Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Žan Palčič

Programiranje vezij FPGA z ogrodjem OpenCL

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: izr. prof. dr. Uroš Lotrič

Ljubljana 2016

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Vezja, ki združujejo klasični procesor in programirljivi čip FPGA, predstavljajo zanimiv heterogeni računalniški sistem, ki je uporaben v aplikacijah, kjer so zahtevane hitrost, zmogljivost in ob enem nizka poraba električne energije. Dodaten zalet daje takim heterogenim sistemom ogrodje OpenCL, s katerim lahko na enak način programiramo oba dela vezja. Za vezje Altera DE1-SoC raziščite kako učinkovita je uporaba ogrodja OpenCL v primerjavi z ostalimi možnimi pristopi. Za nekaj izbranih algoritmov primerjajte kompleksnost pristopa, vpliv različnih prilagoditev in učinkovitosti same implementacije. Rezultate tudi primerjate z izvajanjem na grafično procesni enoti.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Žan Palčič sem avtor diplomskega dela z naslovom:

Programiranje vezij FPGA z ogrodjem OpenCL.

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom izr. prof. dr. Uroša Lotriča,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 24. avgusta 2016 Podpis avtorja:

Najprej bi se rad zahvalil svojemu mentorju, izr. prof. dr. Urošu Lotriču, za navdih, prijaznost in strokovno pomoč pri pisanju diplomskega dela.

Prav tako se za vso podporo, ki sem je bil deležen v vseh letih šolanja, zahvaljujem svoji družini, Maši pa za pogovore, potrpežljivost in vzpodbudo.

Hvala lektorici Mateji Dermelj za prijaznost in lekturo diplomskega dela.

Kazalo

Seznam uporabljenih kratic

kratica	**angleško**	**slovensko**
CPE	Central processing unit	Centralna procesna enota
GPE	Graphics processing unit	Grafična procesna enota
FPGA	Field-programmable gate array	Programirljivo vezje
VHDL	VHSIC Hardware Description Language	Programski jezik za opisovanje strojne opreme
HPS	Hard Processor System	Sistem s centralno procesno enoto
FLOPS	Floating-point operations per second	Število operacij v plavajoči vejici na sekundo
GFLOPS	10^6 FLOPS	10^6 FLOPS

Povzetek

V diplomskem delu se osredotočamo na testiranje programirljivega vezja s pomočjo programskega ogrodja OpenCL in predstavimo različne optimizacije, ki jih ponuja Alterina razširitev ogrodja OpenCL, lastnosti in načine uporabe ploščice FPGA. Naš cilj v delu je bil ugotoviti, kakšna je učinkovitost sinteze vezja z ogrodjem OpenCL, kakšen je vpliv različnih optimizacij na učinkovitost izvajanja ščepcev, in rezultate primerjati z izvajanjem na grafično procesni enoti. Uporabo ogrodja OpenCL na vezju FPGA smo analizirali z implementacijo nenatančnega množilnika, matričnega množenja, Sobelovega filtra in rezanja šivov. To smo implementirali v splošnejši obliki, za izvajanje na več arhitekturah, in v optimizirani obliki, za izvajanje le na vezjih FPGA.

Ključne besede: OpenCL, FPGA, VHDL, heterogeni sistemi, rezanje šivov, Sobelov filter.

Abstract

Main focus of our thesis is testing FPGA circuit with OpenCL framework. The thesis presents different optimization methods that extends OpenCL framework, features of FPGA board, and overview of different programming designs with FPGA board. Our aim is to determine the OpenCL compiler's efficiency while translating high-level kernels to low-level circuit, impact of various optimizations on kernel's runtime and compare results with performance on the graphics processing unit. We analyse designs build with OpenCL on FPGA circuit through implementations of approximate multiplier, matrix multiplication, Sobel filter, and Seam carving. Mainly, the implementations are device independent, for executing kernels on many different architectures, while some of them are optimized for FPGAs only.

Keywords: OpenCL, FPGA, VHDL, Heterogeneous systems, Seam carving, Sobel filter.

Poglavje 1

Uvod

Že dolgo časa so v ospredju večjedrni procesorji in specializirane grafične enote, ki so zmogljivost naših računalnikov občutno povečali. Sistem, ki vsebuje različne procesne enote, imenujemo heterogen sistem. Izmed vseh procesnih enot so se grafično procesne enote (GPE) izkazale za zelo učinkovite pri grafično zahtevnih aplikacijah in tudi pri reševanju problemov z visoko stopnjo podatkovnega paralelizma. Danes si resnično težko predstavljamo računalnik brez več jedrnega procesorja in GPE.

Pri izkoriščanju zmogljivih naprav v heterogenih sistemih, za razvoj aplikacij za znanstvene, finančne namene ali zgolj aplikacij za zabavo, računalniških iger, pogosto uporabljamo programsko orodje OpenCL, ki ga je leta 2009 predstavilo računalniško podjetje Apple. Najbolj razširjeno programsko orodje za izkoriščanje naprav v heterogenih sistemih je postalo zaradi široke podpore naprav različnih proizvajalcev, neodvisnosti od operacijskih sistemov, preproste uporabe, neodvisnosti od vrste pomnilnikov in enostavne prenosljivosti med različnimi arhitekturami. Poleg tega pa je standard OpenCL odprtokoden. Danes za razvoj in standardizacijo OpenCL skrbi skupina razvijalcev Khronos Group.

Zelo razširjene procesne enote GPE, ki so v zadnjih dveh desetletjih med pospeševalniki v računalnikih prevladovale, porabijo veliko električne energije, zato v zadnjem času ponovno prihaja do zanimanja za programirljiva

vezja (angl. field programmable gate array, FPGA). Čeprav so bila vezja FPGA prvotno namenjena za razvoj specifično integriranih vezij, so v zadnjih dveh desetletjih postala zmogljivejša, cenovno dostopnejša in z razvojem sistemov na čipu tudi bolj splošno namenska. Razvijanje aplikacij je z jeziki za opisovanje vezij (angl. hardware description language, HDL) navadno zamudno in predvsem drugačno od običajnega poteka razvijanja aplikacij. Zaradi omenjenih težav je proizvajalec Altera, eden izmed največjih proizvajalcev vezij FPGA, leta 2001 predstavil prvo ploščico oziroma programirljivo vezje s podporo standarda OpenCL. Tako so zahtevnejši razvoj aplikacij z uporabo programskih jezikov za opis vezij FPGA približali tudi programerjem, ki teh jezikov ne znajo uporabljati, so pa seznanjeni s programskim jezikom C in si želijo izkoristiti vse funkcionalnosti in zmogljivosti vezij FPGA.

V številnih člankih [1–3] lahko naletimo na zapise o uporabi in testiranju vezij FPGA in o uporabi programskega orodja OpenCL; v njih so pokazali mnogokratno pohitritev pri enaki porabi električne energije v primerjavi z drugimi procesnimi enotami, kot sta GPE in CPE.

V sklopu diplomske naloge smo se osredotočili na testiranje ploščice FPGA Atera DE1-SoC, ki povezuje vezje FPGA in sistem s procesorjem. Ploščica spada med predstavnike vezij FPGA, ki sodijo v nižji cenovni razred, kljub temu pa ponuja širok nabor vhodno/izhodnih priključkov, sistem na čipu z dvojedrnim procesorjem in zmogljivo vezje FPGA Cyclone V. Želeli smo testirati zmogljivost vezja FPGA z uporabo programskega orodja OpenCL, ga primerjati z uporabo na GPE in spoznati prilagoditve, potrebne za boljše izvajanje ščepcev na vezju FPGA.

V naslednjih poglavjih bomo predstavili tehnologije, ki smo jih uporabili, od heterogenih sistemov, vezja FPGA, jezika za opisovanje vezij VHDL do programskega ogrodja OpenCL, podrobneje pa bomo opisali sestavo ploščice DE1-SoC. Za potrebe testiranja smo razvili pet algoritmov. Implementirali smo nenatančni množilnik, kjer smo se osredotočili na primerjavo med implementacijo v jeziku VHDL in implementacijo s pomočjo ščepcev in programskega ogrodja OpenCL. Pri matričnem množenju smo primerjali izvajanje

z različnimi implementacijami in poskušali algoritem prilagoditi arhitekturi vezja FPGA. Primerjali smo tudi različne optimizacije in prilagoditve, ki jih ponuja Alterina razširitev programskega ogrodja OpenCL. Implementirali smo Sobelov filter in algoritem rezanja šivov, ki sta predstavljala večji problem, omogočala pa sta testiranje izvajanja več ščepcev hkrati. Testirali smo še implementacijo s pomikalnim registrom, ki naj bi se učinkovito sintetizirala na vezje FPGA, in s kanali, ki naj bi omogočali hitrejšo komunikacijo med različnimi ščepci.

Poglavje 2

Heterogeni sistemi

Poleg centralne procesne enote (CPE) imamo v računalnikih tudi specializirane procesne enote, kakršne so na primer grafične procesne enote za obdelavo slik in reševanje problemov, ki imajo visoko stopnjo podatkovnega paralelizma. Centralne procesne enote so namenjene predvsem sekvenčnemu izvajanju kode in splošno namenskim aplikacijam. Poleg enot CPE in GPE lahko v sistem združujemo tudi številne druge procesne enote, kot so programirljiva logična vezja (FPGA) in procesor digitalnih signalov (angl. digital signal processor, DSP). Enoten sistem z omenjenimi procesnimi enotami imenujemo heterogen sistem [4].

Heterogen sistem združuje več procesnih enot, ki opravljajo različne funkcionalnosti in so specializirane ter optimizirane za izvajanje posameznih opravil. Z združevanjem enot v enoten sistem pa se heterogeni sistemi srečujejo tudi s težavami, kot so pregrade med programskimi modeli za različne arhitekture, zakasnitve pri komunikaciji med procesnimi enotami, odpravljanje težav pri naslavljanju zaradi pomnilniške hierarhije, odpravljanje nekonsistentnosti ukazov. Uporabniki heterogenih sistemov si želijo tudi enotno računalniško okolje, ki temelji na razvoju računalniških jezikov, ogrodij in aplikacij, ki izkoriščajo vzporednost. Kljub vsem naštetim težavam heterogeni sistemi prinesejo več dobrega kot slabega.

Poraba električne energije je veliko manjša, če zmogljivo računsko enoto,

na primer eno jedro CPE, ki ima visoko frekvenco ure, razdelimo na več posameznih računskih enot in znižamo frekvenco vsake enote. Skupaj bodo porazdeljene enote z nižjo frekvenco izvedle posamezen ukaz enako hitro kot eno jedro le, če lahko vhodne podatke porazdelimo na več enot [5]. Podoben princip je uporabljen pri enotah v heterogenih sistemih, kjer posamezne enote porabijo manj električne energije, delujejo na nižjih frekvencah in so optimizirane za izvajanje določenih operacij. Če operacije in podatke primerno porazdelimo med specializirane enote, bo rezultat na izhodu sistema dobljen hitreje ali enako hitro kot na visoko frekvenčnem enojedrnem procesorju.

Posamezne enote v heterogenih sistemih so optimizirane za izvajanje specifičnih funkcij, na primer vektorskih operacij, in tako dosežejo učinkovito pohitritev brez odvečnih tranzistorjev in programskih poti. Centralno procesna enota, ki je splošno namenska in ima zmožnost izvajanja najrazličnejših funkcij, je zgrajena iz več tranzistorjev in ima večjo zakasnitev pri izvajanju posameznih ukazov.

Povezljivost vseh procesnih enot, energijska varčnost in povečanje zmogljivosti heterogenih sistemov veliko prispevajo k razvoju računalnikov, vprašanje je le, kako prilagoditi programski model in kako učinkovito odpraviti omenjene težave.

2.1 Vezje FPGA

Vezje FPGA je integrirano vezje, ki ga uporabnik oziroma programer poljubno nastavi. Sestavljeno je iz nastavljivih logičnih enot (angl. adaptive logic module, ALM), te pa so običajno sestavljene iz programabilnih preslikovalnih tabel (angl. look-up table, LUT), pomnilnih celic (angl. flip-flop), seštevalnika in multiplekserja [6]. Z nastavljivimi logičnimi enotami je mogoče implementirati poljubno logično funkcijo. Opis le-te, oziroma konfiguracija integriranega vezja FPGA za določeno funkcijo, je na splošno določena z uporabo programskega jezika za opis strojne opreme.

Prvotno so vezja FPGA razvili z namenom, da bi lahko inženirji načr-

tovali specifično integrirano vezje in preko mnogo iteracij dokončno zgradijo zadovoljiv in kakovosten izdelek za kasnejšo množično proizvodnjo. Sama iteracija in načrtovanje na vezju FPGA sta poceni, proizvodnja končnega ožičenega izdelka pa ne. Če bi napako našli med proizvodnjo bi bili stroški ogromni.

Kljub prvotnemu namenu so vezja FPGA danes zelo močno napredovala in nam poleg same načrtovalske vloge ponujajo tudi visoke zmogljivosti, pri tem pa v primerjavi z drugimi pospeševalniki in splošnonamenskimi procesorji porabijo zelo malo električne energije. Trenutna razvojna usmeritev proizvajalcev vezij FPGA je tudi proizvodnja sistemov na čipu (angl. System on a Chip, SoC), ki zaradi povezljivosti splošnonamenskega procesorja z vezjem FPGA omogočajo veliko dinamičnost pri načrtovanju, nizko porabo energije in zanesljivo delovanje. Nekatera večja podjetja so se, predvsem zaradi nizke porabe FPGA, odločila uporabljati vezja FPGA za pospešitev izvajanja poizvedb na strežnikih. Rezultati so bili zelo dobri tudi pri večjih obremenitvah [1].

Kot smo že omenili, se na CPE za izvajanje določenih algoritmov izvede večje število ukazov, pri katerih posamezne enote na CPE niso vedno najbolj izkoriščene. Pri vezju FPGA se z uporabo le nujnih gradnikov in zaradi paralelne enote izognemo dolgim cevovodom in nepotrebnim računskim enotam, ki za naš algoritem niso potrebni. Tako dosežemo boljše časovne zmogljivosti, vendar pa običajno porabimo več fizičnega prostora oziroma komponent kot pri enostavnem procesorju.

Poglavje 3

Izbrana tehnologija

3.1 VHDL

VHDL (angl. Very High Speed Integrated Circuit Hardware Description Language) je programski jezik za opis, modeliranje in sintezo digitalnih vezij. Pri programiranju navadno uporabljamo tri osnovne konstrukte in sicer, programske knjižnice, objekte in arhitekture.

Programske knjižnice vsebujejo že definirane procese in module, ki nam omogočajo hitrejši razvoj aplikacij. Objekt je osnovni konstrukt, ki definira vhodne in izhodne povezave v danem okolju. Deklariramo zunanje signale (angl. port) in jim določimo ime, podatkovni tip in smer. Signali predstavljajo osnovne povezave za opis digitalnega vezja, arhitektura pa opisuje delovanje vezja.

Osnovni elementi pri opisovanju delovanja vezja so, poleg prireditvenih stavkov in pogojnih prireditvenih stavkov, procesi. Znotraj procesov je vrstni red izvajanja stavkov pomemben, mogoče je uporabiti pogojne stavke, procesi pa prav tako vsebujejo pomnilne elemente. Proces se izvrši, ko se spremeni vrednost vsaj enega izmed signalov na katerega je občutljiv. Več procesov, ki so občutljivi na isti signal, se izvaja vzporedno.

Primer uporabe osnovnih elementov si lahko pogledamo v izseku 3.1.

Izsek 3.1: Primer implementacije entitete register z VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity REG is
     port (
      clk_in   :  IN std_logic;
      rst_in   :  IN std_logic;
      data_in  :  IN std_logic_vector(31 downto 0);
      data_out :  OUT std_logic_vector(31 downto 0)
    );
end entity;
architecture Behavioral of REG is
   signal q : std_logic_vector(31 downto 0);
begin
   process (clk_in)
   begin
      if clk_in='1' and clk_in'event then
         if rst_in='1' then
            q <= (others => '0');
         else
            q <= data_in;
         end if;
      end if;
   end process;
   data_out <= q;
end;
```

S programskim jezikom VHDL in različnimi programskimi okolji lahko s pomočjo simulacije preverimo delovanje opisanega vezja. Na koncu se s logika s sintezo pretvori na nivo logičnih vrat in pomnilnih celic.

3.2 OpenCL

OpenCL je programsko ogrodje za programiranje in izvajanje programov na heterogenih sistemih. Sestavljajo ga programski jezik OpenCL C, ki temelji na standardu ISO C99, programski vmesnik (angl. OpenCL API), ki skrbi za nadzor platform in izvajanje programov na določenih procesnih enotah, knjižnice in gonilniki za razvoj programske opreme. Kljub temu da program-

ski jezik OpenCL C temelji na standardu ISO C99, je pri programiranju nekaj omejitev, in sicer ne podpira rekurzije, kazalcev na funkcije, polj in struktur spremenljive dolžine [7].

Ogrodje OpenCL lahko razdelimo na štiri modele:

- model okolja – visoko-nivojski prikaz heterogenega sistema in vseh naprav v sistemu;
- model izvajanja – predstavitev poteka ukazov na napravah;
- pomnilniški model – predstavitev pomnilniške hierarhije znotraj ogrodja in interakcije med posameznimi nivoji;
- programski model – abstrakcija, potrebna za implementiranje in izvajanje ščepcev v različnih načinih.

V ogrodju OpenCL imamo vedno enega gostitelja, na katerega je povezana ena ali več računskih naprav. Računska naprava je lahko CPE, GPE, DSP, FPGA oziroma katerakoli procesna enota, ki ima podporo za programsko ogrodje OpenCL. Posamezna naprava je razdeljena na več računskih enot, te pa na več procesnih elementov.

Program je v ogrodju OpenCL sestavljen iz gostiteljskega programa in enega ali več ščepcev (angl. kernel). Gostiteljski program se izvaja na gostitelju in komunicira z računskimi napravami, na katerih se izvajajo ščepci. Ščepci so navadno preproste podatkovno paralelne funkcije, napisane v programskem jeziku OpenCL C. Definirajo delo posamezne delovne enote ali delavca (angl. work-item). Vsako nit, ki izvaja ščepec, imenujemo delavec. Vsak delavec ima svojo globalno identifikacijsko število (angl. global ID), ki ga enolično označuje v globalnem razponu. Globalni razpon je predstavljen prostor v N-dimenzionalnem razponu, ki je lahko najmanj eno- in največ tridimenzionalen. Več delavcev skupaj predstavlja delovno skupino (angl. work-group), znotraj katere so delavci enolično določeni z lokalno identifikacijsko številko (angl. local ID). Znotraj delovne skupine se lahko delavci

sinhronizirajo in si med seboj delijo lokalni pomnilnik. Poleg globalne in lokalne identifikacijske številke ima vsak delavec tudi številko delovne skupine, ki ji pripada.

OpenCL definira štirinivojsko pomnilniško arhitekturo:

- globalni pomnilnik – dosegljiv je z vseh procesnih enot, ima visoko latenco in ni sinhroniziran;
- pomnilnik konstant – del globalnega pomnilnika, do katerega je mogoč samo bralni dostop, se med izvajanjem ne spreminja;
- lokalni pomnilnik – skupni pomnilnik znotraj delovne skupine;
- zasebni pomnilnik – registri za posameznega delavca.

Zakasnitev dostopa do pomnilnika pri naštetih nivojih pada.

Programsko ogrodje OpenCL definira tudi dva različna programska modela: opravilni in podatkovni paralelizem. Pri podatkovnem paralelizmu so ključne nastavitev problemskega razpona, velikost delovnih skupin in prilagoditev izvajanje ščepcev gleda na podatkovne strukture. Podatki se porazdelijo med delavce v delovnem razponu. Prvotno je ogrodje usmerjeno v podatkovni paralelizem, omogoča pa tudi opravilni paralelizem. Ta omogoča izvajanje več ščepcev hkrati.

Ena ključnih lastnosti programskega ogrodja je prenosljivost, vendar pa kljub dobri podpori za prenosljivost programov ni zagotovila, da bo med različnimi pospeševalniki učinkovitost izvajanja enaka. Za optimalno izvajanje paralelnega dela potrebuje vsaka procesna enota oziroma vsak pospeševalnik ustrezne prilagoditve svoji arhitekturi.

3.3 Programski jezik C++

Programski jezik C++ je splošno namenski računalniški programski jezik, ki poleg proceduralnega programerskega pristopa omogoča objekten pristop.

Pri razvoju gostiteljskega programa v programskem okolju OpenCL obstajajo ovojnice, ki omogočajo objektno programiranje in lažje definiranje okolja OpenCL na gostitelju z objekti ter tako boljšo prenosljivost definiranega okolja med različnimi problemi. Poleg objektnega pristopa pa se programski jezik C++ prevede v strojni jezik in tako je veliko hitrejši od jezikov, ki se interpretirajo. Omogoča tudi zelo nizkonivojski dostop do pomnilnika. Za potrebe testiranja algoritmov na grafični kartici AMD HD7870 smo uporabili prevajalnik Visual C++. Gostiteljski program za testiranje na ploščici FPGA, natančneje za sistem s centralno procesno enoto (angl. hard processor system, HPS) oziroma procesor ARM, smo prevedli z ukazom, prikazanim v izseku 3.2.

Izsek 3.2: Primer ukaza za prevajanje gostitelja za procesor ARM

```
arm-linux-gnueabihf-g++ host/src/Main.cpp host/src/
    AlteraOpenCl.cpp <ostale_vhodne_datoteke> -o
    MatrixMultiplierHost -IC:/Altera/15.0/hld/host/
    include -I../ -Ihost/inc -I../../../ -LC:\Altera
    \15.0\hld\board\terasic\de1soc\arm32\lib -LC:/
    Altera/15.0/hld/host/arm32/lib -Wl,--no-as-needed
    -lalteracl -lalterahalmmd -lalterammdpcie -lelf -
    lrt -ldl -lstdc++
```

3.4 Izbrana strojna oprema

3.4.1 AMD HD7870

AMD HD7870 je zmogljiva GPE, ki je pri računanju v enojni natančnosti precej učinkovita. GPE ima 1280 jeder v 20 procesnih enotah. Frekvenca ure na posameznih jedrih je 1,1 GHz, pri prenosu podatkov pa 1,2 GHz. Največja teoretična zmogljivost pri enojni natančnosti je 2560 GFLOPS, pri dvojni pa le 160 GFLOPS. Grafično procesna enota ima na voljo tudi 2 GB pomnilnika GDDR5 in je preko 256 bitnega spominskega vodila povezana na

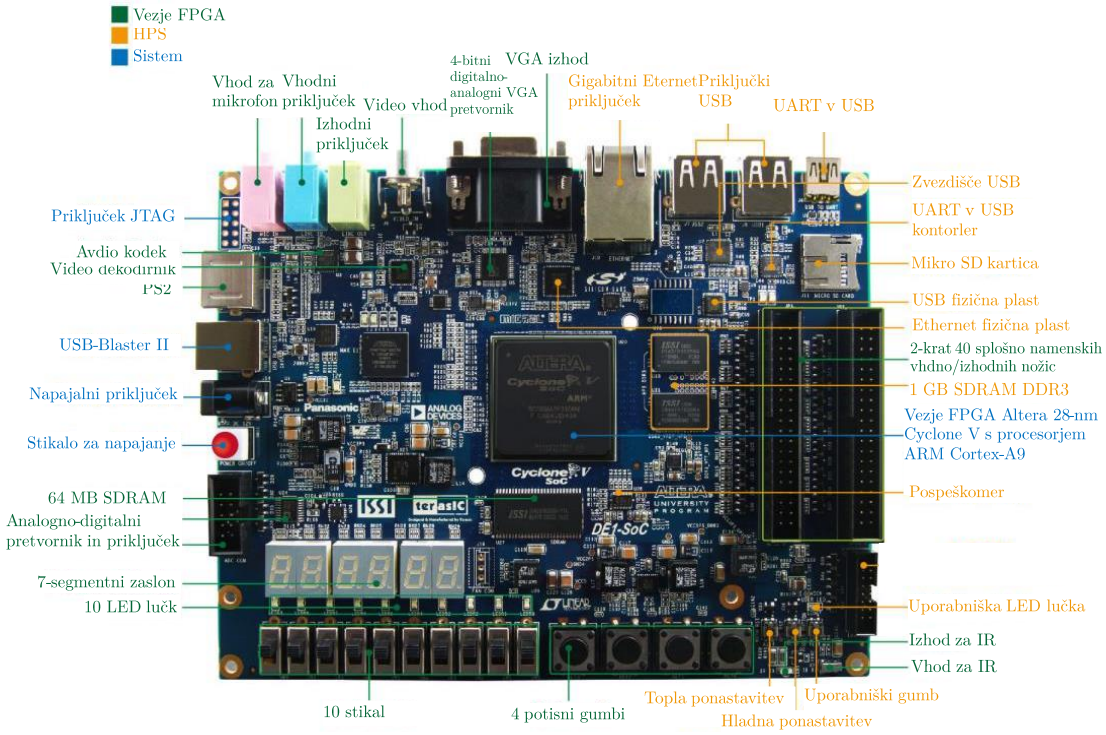
gostitelja. Največja teoretična hitrost prenosa podatkov preko vodila je 153,3 GB/s. Pri polni obremenitvi potrebuje približno 275 W električne moči [8].

3.4.2 Plošča Altera DE1-SoC

Plošča Altera DE1-SoC predstavlja robustno zasnovano strojno opremo, ki so jo razvili na osnovi sistema na čipu FPGA. Združuje visoko zmogljiv dvojedrni procesor ARM Cortex-A9 z nizko porabo in vezje FPGA, ki uporabniku omogoča fleksibilnost [9]. Poleg tega ponuja velik nabor vhodno/izhodnih vmesnikov za povezovanje številnih naprav. Povezavo med sistemom s centralno procesno enoto, ki ga sestavljajo zmogljiv procesor, spomin in vhodno/izhodni vmesniki, in vezjem FPGA omogoča napredno razširljiv vmesnik (angl. advanced extensible interface, AXI).

Sistem HPS in vezje FPGA si ne delita vseh zunanjih priključkov. Določeni zunanji vmesniki so dodeljeni le FPGA, drugi pa sistemu HPS. Te vmesnike nastavimo s pomočjo zagonskega programa na večprocesorskem sistemu, vmesnike na vezju FPGA pa preko slike na sistemu HPS ali preko drugih kompatibilnih zunanjih virov. Nastavitev vezja FPGA preko zunanjih virov, delovnega računalnika, na katerega je priključena ploščica, imenujemo tudi programski model JTAG. Nastavitve se ob ponovnem zagonu ploščice izgubijo. Natančnejša razdelitev posameznih komponent med sistemom HPS in vezjem FPGA je prikazana na sliki 3.1.

Sistem HPS in FPGA imata ločena vira napajanja, vendar je sistem HPS konstantno prižgan, vezje FPGA pa je lahko prižgano ali ugasnjeno.



Slika 3.1: Pogled na ploščo od zgoraj in razdelitev komponent med podsistema HPS in FPGA

FPGA

Na plošči DE1-SoC vezje FPGA povezuje naslednje pomembne komponente [9]:

- Altera Cyclone V SE5CSEMA5F31C6N;
- USB-Blaster II – omogoča programiranje na plošči (način JTAG);
- 64MB SDRAM (16-bitno podatkovno vodilo);
- štiri potisne gumbe;
- deset stikal;
- deset rdečih lučk LED;

- šest 7-segmentih zaslonov;
- štiri 50MHz ure;
- priključek PS/2 za miško ali tipkovnico.

Vezje FPGA Cyclone V je zgrajeno na osnovi 28 nm tehnologije. Posamezna logična enota je sestavljena iz 8-vhodnega LUT, dveh polnih seštevalnikov, multiplekserjev in štirih namenskih registrov, ki pripomorejo k optimizaciji izvajanja.

Ploščica Altera DE1-SoC ima vezje FPGA Altera Cyclone V z oznako SE 5CSEMA5F31C6N. Število posameznih gradnikov si lahko ogledamo v tabeli 3.1.

Tabela 3.1: Gradniki FPGA

Ime gradnika	Število ali velikost
Logični element	85000
ALM	32075
Register	128300
Spomin (kB)	4450
Spominski blok	397
DSP blok	87
18x18 multiplekser	174

HPS

Sistem s centralno procesno enoto je zgrajen iz:

- 800MHz dvojedernega procesorja ARM Cortex-A9;
- 1GB pomnilnika DDR3 SDRAM (32-bitno podatkovno vodilo);
- gigabitnega priključka Ethernet;
- reže micro SD;
- priključka UART v USB, USB mini.

Ostale komponente so navedene v uporabniških navodilih [9].

Most HPS-FPGA AXI

Most HPS-FPGA podpira napredno razširljiv vmesnik (angl. Advanced eXtensible Interface, AXI) in je sestavljen iz treh različnih elementov [6].

- Element FPGA-to-HPS v mostu AXI predstavlja visoko zmogljiv vmesnik, ki podpira 32-, 64- in 128-bitno podatkovno širino in omogoča vezju FPGA prenos podatkov na HPS.

- Element HPS-to-FPGA v mostu AXI predstavlja visoko zmogljiv vmesnik, ki podpira 32-, 64- in 128-bitno podatkovno širino in omogoča prenos podatkov HPS na vezje FPGA.

- Element HPS-to-FPGA v lahkem mostu AXI pa omogoča le 32-bitno podatkovno širino in prenos podatkov iz sistema HPS na vezje FPGA. Običajno se ta model uporablja le za komunikacijo z vhodno/izhodnimi napravami in za dostop do statusnih registrov.

Poleg omenjenih povezav ima plošča na voljo kontrolni sistem HPS SDRAM, ki je sestavljeno iz večvratnega krmilnika SDRAM in vmesniškega protokola, ki definira povezovanje med krmilniki DDR in pomnilnimi komponentami (angl. DDR physical layer interface, DDR PHY). Kontrolni sistem HPS SDRAM si delijo predpomnilnik L2, vezje FPGA, ki dostopa do SDRAM preko vmesnika FPGA-to-HPS SDRAM, in predpomnilnik L3. Vmesnik FPGA-to-HPS SDRAM je tudi privzeti način komunikacije med vezjem FPGA in sistemom HPS.

Običajen potek načrtovanja

Pri načrtovanju projekta s ploščico DE1-SoC je treba najprej določiti, kako bomo ploščico uporabili. Prvi način je samo uporaba vezja FPGA, drugi

uporaba večnamenskega procesorja ARM na sistemu HPS, tretji pa je kombinacija obeh.

Pri prvem načinu razvoja aplikacije za vezje FPGA uporabimo programsko orodje DE1-SoC System builder, ki nam omogoča izbor potrebnih vhodno/izhodnih nožic. Po izboru vseh potrebnih nožic ustvarimo datoteko v strojno opisnem jeziku Verilog ali VHDL in nastavitveno datoteko Quartus II. Ta vsebuje prireditev nožic in nastavitve parametrov za posamezne nožice. Datoteko Verilog/VHDL uporabnik dopolni s svojo poljubno uporabniško logiko in temu primerno doda svoje strojno opisne datoteke. Na koncu je treba projekt prevesti in nastaviti vezje FPGA na ploščici z nastavitveno datoteko, ki ima končnico ".SOF". Nastavitev vezja FPGA je potrebna ob vsakem zagonu sistema, če nastavitven datoteka ni nastavljena kot privzeti način zagona vezja FPGA [10].

Pri drugem načinu, pri katerem želimo načrtovati program za sistem HPS, je treba program napisati v programskem jeziku C s poljubnim urejevalnikom besedil, kodo prevesti z ustreznim prevajalnikom (Altera SOC EDS), zagnati operacijski sistem Linux iz spominske kartice na ploščici DE1-SoC, kopirati izvršljivo datoteko na ploščico in jo izvršiti [11].

Pri tretjem načinu, pri katerem program uporablja sistem HPS kot vezje FPGA, je treba ustvariti projekt s programskim orodjem Quartus II, podobno kot pri prvem načinu, le da tu poleg uporabniške logike v strojnem opisnem jeziku povežemo še vhodno/izhodne komponente s sistemom HPS. To storimo s pomočjo vgrajenega orodja Qsys. Po vseh naštetih korakih program še prevedemo z orodjem Quartus II in izvršimo posebno namensko skripto za ustvarjanje zaglavnih datotek, ki so potrebne pri pisanju programa za sistem HPS. V zaglavni datoteki so predvsem definirani bazni naslovi, preko katerih lahko dostopamo do vhodno/izhodnih komponent, ki smo jih predhodno povezali z orodjem Qsys. Sledi programiranje uporabniške logike za sistem HPS v programskem jeziku C, kot smo omenili pri drugem načinu. Dodatno je potrebna le uporaba funkcij za preslikovanje naslovov in dostopov do naslovljenega spomina, ki so podrobneje opisani v uporabniških navodi-

lih [9, 12]. Vezje FPGA je treba nastaviti z nastavitveno datoteko ".SOF"in nato izvršiti izvršljivo datoteko, ki smo jo dobili s prevajanjem programa v programskem jeziku C [12].

Pri tretjem načinu, pri katerem uporabljamo sistem HPS in vezje FPGA, je s prevajalnikom Altera OpenCL SDK mogoče tudi implementiranje vzporednih algoritmov za FPGA preko vmesnika OpenCL. To nam omogoča hitrejši razvoj aplikacij, saj nam ni treba ločeno razvijati logike s programskimi jeziki HDL za vezje FPGA in posebej logike za sistem HPS s programskim jezikom C. V nadaljevanju bomo preverili učinkovitost izvajanja algoritmov, implementiranih s pomočjo programskega vmesnika OpenCL, na vezju FPGA.

Altera OpenCL SDK in prevajanje ščepcev

Pri prevajanju programske kode je treba prevesti sekvenčni del kode s standardnim prevajalnikom arm-linux-gnueabihf-g++ in posebej prevesti ščepec s prevajalnikom Altera offline Compiler. Rezultat prevajanja sekvenčne kode je izvršljiv program na gostitelju, sistemu HPS. Pri prevajanju ščepca se po oceni porabe gradnikov na vezju FPGA ustvari vmesna datoteka v programskem jeziku HDL, ki je nato posredovana prevajalniku za HDL jezike Quartus II. Rezultat končnega prevajanja je izvršljiva datoteka, ki jo v času izvajanja gostiteljskega programa izvede gostitelj na FPGA (glej sliko 3.1). Prevajanje in sintetiziranje kode HDL v prevajalniku Quartus II je skrito v ozadju prevajalnika Altera OpenCL [13].

Ukaz, ki smo ga uporabili za prevajanje ščepca brez argumentov, je naslednji:

```
aoc <ime_scepca>.cl -o <ime_izvrsljivega_scepca>.aocx
```

.

Pri prevajanju smo uporabili tudi naslednje argumente, ki so pripomogli k optimizaciji in razhroščevanju programske kode [13, 14]:

–report

Prevajalnik pri prevajanju oceni porabo različnih virov oziroma gra-

dnikov na ploščici za realizacijo ščepca. Z uporabo tega argumenta jih prikaže na zaslonu. Oceni relativno porabo vseh logičnih elementov, spominskih blokov, blokov DSP in registrov, ki so na voljo za določeno ploščo. Za oceno porabe logičnih gradnikov porabi malo časa. Prevajalnik ne zaupa svoji prvi oceni in tako, kljub ogromnemu presežku potrebnih gradnikov za realizacijo, poskuša optimizirati programske poti in zmanjšati število potrebnih gradnikov. Po številnih iteracijah neuspešnega optimiziranja opozori, da mu logike za dano vezje ni uspelo realizirati. Z uporabo tega argumenta prihranimo veliko časa pri implementaciji ščepcev, saj se izognemo nepotrebnemu prevajanju.

–profile

Ta argument ščepcu doda programske oziroma zmogljivostne števce, ki v času izvajanja ščepca na FPGA merijo zakasnitve v cevovodu, zakasnitve med kanali, ki povezujejo ščepce, in hitrost prenosa do globalnega pomnilnika. Pri izvajanju ščepcev se na koncu ustvari datoteka profile.mon, ki nam omogoča vpogled v tako imenovana ozka podatkovna grla, in optimizacijo kode v naslednjih iteracijah. Seveda števci obremenijo učinkovitost izvajanja, zato jih uporabimo, le ko kodo optimiziramo.

–no-interleaving default

Argument nam onemogoči privzeti način prenosa podatkov iz globalnega pomnilnika, ki naj bi bil optimiziran za najrazličnejše primere. Globalni pomnilnik je razdeljen na več manjših blokov in z nihajočim oziroma z izmeničnim prenosom med bloki optimizira hitrost prenosa. Vendar to, odvisno od problema, ni vedno optimalno, zato je mogoča tudi ročna razdelitev globalnega pomnilnika v večje ali manjše bloke, s čimer je optimiziran dostop do globalnega pomnilnika. Pri tem moramo v gostiteljskem programu pri ustvarjanju pomnilnika uporabiti ustrezno zastavico (CL_MEM_BANK_<stevilo_bloka >_ALTERA).

–fp-relaxed

Pri računanju z aritmetičnimi operacijami prevajalnik navadno sestavi dolgo cevovodno obliko in v določenem vrstnem redu izvaja operacije. Dolgi cevovodi se izvajajo tudi več urinih period, zato si želimo izvajanje pohitriti z optimalnejšim vrstnim redom izvajanja aritmetičnih operacij. S podanim argumentom –fp-relaxed prevajalniku povemo, da hočemo, da vse operacije izvede bolj enostavno in ne tako striktno. Namesto dolgega cevovoda sintetizira uravnoteženo drevo in tako oblikuje širšo cevovodno obliko, ki izboljša učinkovitost izvajanja. Pri tem ima lahko rezultat manjšo napako.

Z uporabo programskega vmesnika Altera SDK za programsko okolje OpenCL se ščepci prevedejo v visoko paralelno vezje. Za vsako operacijo v ščepcu je narejena unikatna funkcionalna enota, pri tem pa se različne funkcionalne enote povezujejo. Poleg paralelnih funkcionalnih enot pa je izkoriščen tudi cevovodni paralelizem, ki poskrbi, da vezje vsako urino periodo ohranja funkcijske enote zaposlene.

Komunikacija med gostiteljem in vezjem FPGA na plošči FPGA SoC poteka zelo hitro, saj si oba bloka delita skupni naslovni prostor. Za posredovanje podatkov ščepcu ni potrebno dodatno pošiljanje iz gostitelja na pospeševalnik in tudi zakasnitev pri dostopu je veliko manjša. Med izvajanjem gostitelj locira pomnilnik za FPGA, na FPGA se izvede ščepec oz. več ščepcev, rezultati se izračunajo in zapišejo v pomnilnik. Kot smo omenili v poglavju 3.4.2, se podatki privzeto prenašajo preko vmesnika FPGA-to-HPS SDRAM, ki predstavlja globalni naslovni prostor v ščepcu.

Poglavje 4

Implementacija

4.1 5-bitni množilnik

Za površinski pregled delovanja ploščice smo implementirali množilnik 5-bitnih števil. Začeli smo implementacijo množilnika z uporabo vezja FPGA in temu ustrezno povezali vhodno/izhodne nožice. Napisali smo uporabniško logiko množilnika v jeziku VHDL in povezali vhode na stikala in gumbe. Množilnik je ob pritisku na določen gumb prebral dve 5-bitni števili preko stikal, izračunal produkt in ustrezno prižgal diode LED, ki so prikazale rezultat množenja.

Enak primer množilnika smo implementirali tudi z načinom FPGA-HPS, kjer smo vhodno/izhodne komponente FPGA povezali s sistemom HPS. Uporabniško logiko množilnika smo implementirali s programskim jezikom C, jo prevedli in izvršljivo datoteko izvedli na sistemu HPS.

4.2 Nenatančni množilnik

Implementacija nenatančnega množilnika temelji na osnovi dveh člankov [15, 16]. Osnovna ideja nenatančnega množilnika je predvsem predstavitev števil v logaritemskem zapisu in v približku njunega produkta.

Produkt dveh pozitivnih celih števil lahko predstavimo kot vsoto

$$\log_2 (N_1 \cdot N_2) = \log_2 N_1 + \log_2 N_2 \quad .$$

Če nastavimo $k_1 = \lfloor \log_2 N_1 \rfloor$ in $k_2 = \lfloor \log_2 N_2 \rfloor$, potem velja, da je logaritem produkta približno enak $\log_2 (N_1 \cdot N_2) \approx k_1 + k_2$ oziroma da je produkt dveh števil približno enak

$$N_1 \cdot N_2 \approx 2^{k_1+k_2} \quad .$$

Po tem moramo upoštevati napako pri izračunu produkta, in sicer lahko število N predstavimo kot $N = 2^k + N^{ost}$, pri čemer k označuje vodilno enico v bitnem zapisu in N^{ost} ostanek po odstranitvi vodilne enice. Tako lahko natančen produkt zapišemo kot

$$\begin{aligned} N_1 \cdot N_2 &= (2^{k_1} + N_1^{ost}) \cdot (2^{k_2} + N_2^{ost}) \\ &= 2^{k_1+k_2} + 2^{k_1} \cdot N_2^{ost} + 2^{k_2} \cdot N_1^{ost} + N_1^{ost} \cdot N_2^{ost} \quad . \end{aligned}$$

Del natančnega produkta oziroma prvi približek produkta $P^0_{priblizek} = 2^{k_1+k_2} + 2^{k_1} \cdot N_2^{ost} + 2^{k_2} \cdot N_1^{ost}$ lahko izračunamo z operacijo seštevanja in pomikalnim registrom, ki se na vezju FPGA izvede zelo hitro. Za absolutno napako oziroma ostanek $E^0 = N_1^{ost} \cdot N_2^{ost}$, $E^0 > 0$ pa lahko zmnožimo v naslednji iteraciji na enak način, ki je opisan zgoraj (4.2), in zapišemo produkt kot $E^0 = C^1 + E^1$, kjer C^1 označuje približek produkta in E^1 absolutno napako. Tako lahko natančen produkt definiramo kot

$$P_{natancen} = P^0_{priblizek} + C^1 + E^1$$

oziroma če postopek ponavljamo več iteracij, lahko približek produkta posplošimo kot

$$P^i_{priblizek} = P^0_{priblizek} + \sum_{j=1}^{i} C^j \quad .$$

4.2.1 Implementacija z VHDL

Pri implementaciji z VHDL smo sestavili več osnovnih entitet, ki smo jih uporabili. Entiteta približni množilnik med seboj povezuje entitete, imenovane osnovni blok, povezave s stikali in gumbi in uro na vezju FPGA. Entiteta osnovni blok izračuna eno iteracijo produkta po metodi, opisani v poglavju 4.2. Osnovni blok je razdeljen na štiri stopnje in prve tri stopnje vmesni rezultat shranijo v entiteto register. Poleg entitete register imamo implementirana še entiteto za določanja položaja vodilne enice (angl. Leading One Detector, LOD) in pomikalni register.

V prvi stopnji osnovnega bloka vsakemu faktorju s pomočjo entitete LOD in z ekskluzivno disjunkcijo določimo položaj vodilne enice, k_1 oziroma k_2, in ostanek števila brez vodilne enice. Če imamo več osnovnih blokov, ostanek števila brez vodilne enice posredujemo naslednjemu bloku, kot faktor. V drugi stopnji seštejemo lokaciji obeh vodilnih enic in s pomikalnim registrom pomaknemo ostanek prvega faktorja za k_2 in ostanek drugega faktorja za k_1 mest v levo. V tretji stopnji cevovoda seštejemo rezultata iz pomikalnih registrov $N_2^{ost} \cdot 2^{k_1} + N_1^{ost} \cdot 2^{k_2}$ in dekodiramo vsoto vodilnih enic v število $2^{k_1+k_2}$. V zadnji stopnji seštejemo še vmesna rezultata iz stopnje tri in tako dobimo $P^0_{priblizek}$, ki ga nato prištejemo k naslednjim približkom, če so ti prisotni.

4.2.2 Implementacija z OpenCL

Pri implementaciji s ščepci uporabimo več delavcev in tako poskušamo doseči vzporedno računanje osnovnih blokov in se s pomočjo sinhronizacije delavcev približati cevovodu, ki smo ga implementirali v VHDL.

Vsak delavec dobi en faktor in gre skozi vse stopnje cevovoda. Če uporabimo več osnovnih blokov, se delavci počakajo, dokler nimajo vsi svojih faktorjev. Delavec nato za svoj faktor izračuna lokacijo vodilne enice, ostanek in vmesne rezultate s pomikalnim registrom. Na koncu prvi delavec sešteje približke in vrne rezultat gostitelju.

Implementacija ščepca z enim delavcev izračuna vse potrebne vmesne rezultate za vsak faktor in nato približni produkt vrne gostitelju. S to implementacijo želimo testirati, ali se cevovodna oblika izkaže za učinkovitejšo kot implementacija z več delavci.

4.3 Matrično množenje

Matrično množenje je matematična operacija, ki zmnoži dve matriki in ustvari novo matriko [17]. Če imamo matriko A velikosti $m \times n$, kjer m predstavlja število vrstic ali višino matrike in n število stolpcev ali širino matrike z elementi matrike a_{ik}, $1 \leq i \leq m$ in $1 \leq k \leq n$

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} ,$$

in matriko B velikosti $n \times p$, z elementi b_{kj}, $1 \leq k \leq n$ in $1 \leq j \leq p$

$$B = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix} ,$$

potem je produkt matrik enak $C = A \times B$ velikosti $m \times p$

$$C = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix} ,$$

z elementi

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \quad .$$

4.3.1 Implementacija algoritma

Vrstična implementacija

Pri prvi implementaciji matričnega množenja je osnovna ideja, da posamezna nit izračuna celotno vrstico matrike C tako, da za vsak element v vrstici matrike C sešteje produkte med vsemi elementi dane vrstice v matriki A z vsemi elementi v ustreznem stolpcu matrike B. Posamezna nit uporablja za izračun elementa c_{ij} elemente a_{ij}, $i = 1 \ldots n$ in iterira skozi stolpce matrike B z elementi b_{ij}, $j = 1 \ldots n$. Vsaka nit ima tako natančno določen i, ki je enak globalni identifikacijski številki, in indeks $j = 1, 2, 3, \ldots, n$. Pri tem je potreben pogoj, da ustvarimo globalni razpon velikosti m, če pa dopuščamo, da je globalni razpon večji, je zato potreben izključitveni pogoj za niti, ki presegajo mejo m. Za boljšo predstavo implementacije algoritma si lahko pogledamo psevdokodo 1.

Pri algoritmu za izračun posamezne vrstice matrike C posamezna nit za izračun naprimer matrike velikosti $n \times n$ opravi n operacij množenja in n operacij seštevanja, pri tem pa dostopa tako do lokalnega kot do globalnega pomnilnika. Prednost te implementacije je predvsem v uporabi lokalnega pomnilnika, saj si niti v delovni skupini pred začetkom množenja in seštevanja shranijo celoten stolpec matrike B. Ta stolpec je tako uporabljen znotraj celotne skupine niti. Problem, ki se tu pojavi, je, da je največja velikost delovnih skupin navadno omejena na 256 oziroma na precej majhno število d. Tako ima matrika z n vrsticami $\lceil n/d \rceil$ delovnih skupin, pri čemer vsaka v svoj lokalni pomnilnik kopira enake stolpce in tako izvajanje ni optimalno. Rezultate si bomo ogledali v nadaljevanju.

Algoritem 1 Izračun posamezne vrstice matrike C v ščepcu

```
1: gid ← indeks trenutne vrstice
2: l_velikost ← velikost delovne skupine
3: b_stolpec[stevilo_vrstic_B]                ▷ Lokalni pomnilnik za stolpec matrike B
4: i ← 0
5: for i < stevilo_stolpcev_C do
6:     j ← indeks znotraj delovne skupine
7:     for j < stevilo_vrstic_B; j ← j+l_velikost do
8:         B_stolpec[j] ← B[j][i]
9:     end for
10:    pregrada                               ▷ Počakamo na sinhronizacijo niti
11:    produkt ← 0
12:    k ← 0
13:    for k < stevilo_vrstic_B do
14:        produkt ← produkt + A[gid][k] * B_stolpec[k]
15:    end for
16:    C[gid][i] ← produkt                    ▷ Rezultat zapišemo v globalni pomnilnik
17: end for
```

Implementacija s ploščicami

Pri implementaciji matričnega množenja je osnovna ideja, da posamezna nit izračuna en element matrike C. Niti so združene v kvadratne podmatrike oziroma delovne skupine, ki izpolnjujejo potrebna pogoja $0 \equiv m \mod w$ in $0 \equiv p \mod w$, kjer m predstavlja višino matrike A oziroma C, p širino matrike B oziroma C in w širino oziroma višino ploščice, kvadratne podmatrike v matriki C. Pri tem je problem najlažje predstaviti v dvodimenzionalnem globalnem razponu in tako ustvariti $m \times p$ niti. Enostavno povedano, posamezen ščepec izračuna ploščico velikosti $w \times w$ matrike C, tako da v lokalni pomnilnik shrani ploščico iz matrike A in ploščico iz matrike B in izračuna njun prispevek. Ščepec postopek ponavlja dokler ne uporabi vseh potrebnih ploščic. Za boljši pregled nad implementacijo algoritma si lahko pogledamo psevdokodo 2.

Algoritem 2 Izračun posamezne podmatrike ali ploščice matrike C v ščepcu

1: $w \leftarrow$ *indeks trenutnega stolpca*
2: $h \leftarrow$ *indeks trenutne vrstice*
3: $lw \leftarrow$ *lokalni identifikator v 1. dimenziji lokalnega razpona*
4: $lh \leftarrow$ *lokalni identifikator v 2. dimenziji lokalnega razpona*
5: *sirina_ploscice* $\leftarrow$ *velikost delovne skupine v enem lokalnem razponu;*
6: $Al[sirina\_ploscice][sirina\_ploscice]$ ▷ Lokalni pomnilnik za ploščico iz matrike A
7: $Bl[sirina\_ploscice][sirina\_ploscice]$ ▷ Lokalni pomnilnik za ploščico iz matrike B
8: $iA \leftarrow$ *indeks elementa v matriki A, ki predstavlja začetek za določeno skupino delavcev*
9: $iB \leftarrow$ *indeks elementa v matriki B, ki predstavlja začetek za določeno skupino delavcev*
10: *zadnji* $\leftarrow$ *indeks elementa zadnje ploscice v matriki A za določeno skupino delavcev*
11: $produkt \leftarrow 0$
12: **for** $iA <= zadnji; iA = iA + sirina\_ploscice; iB = iB + sirina\_ploscice$ **do**
13: $Al[lh][lw] \leftarrow A[lh][lw + i]$
14: $Bl[lw][lh] \leftarrow B[lh][lw + i]$
15: **pregrada** ▷ Počakamo na sinhronizacijo niti
16: $i \leftarrow 0$
17: **for** $i < sirina\_ploscice$ **do**
18: $produkt \leftarrow produkt + Al[lh][i] * Bl[lw][i]$
19: **end for**
20: **pregrada** ▷ Počakamo na sinhronizacijo niti
21: **end for**
22: $C[h][w] \leftarrow produkt$ ▷ Rezultat zapišemo v globalni pomnilnik

4.4 Sobelov filter

Sobelov filter ali Sobelov operator je matematični postopek za odkrivanje robov na slikah. Postopek uporabljamo pri procesiranju slik, pri katerem operator slikovno poudari robove na sliki. Filter izračuna gradiente po osi x in y s konvolucijo izvorne slike s pomočjo konvolucije jedrc. Približek

gradientov slike I po osi x in y lahko zapišemo kot

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * I, \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * I \quad . \tag{4.1}$$

Pri tem oznaka $*$ predstavlja dvodimenzionalno konvolucijo. Magnitudo nato izračunamo kot

$$G = \sqrt{G_x^2 + G_y^2} \quad ,$$

vendar je v praksi, za hitrejši izračun, običajno dovolj le njen približek, pri čemer izgubimo nekaj informacije [5]

$$G = |G_x| + |G_y| \quad . \tag{4.2}$$

4.4.1 Implementacija algoritma

Pri Sobelovem operatorju smo se problema prav tako lotili na dva načina. Pri prvem smo majhen del slike kopirali v lokalni pomnilnik in nato je vsaka nit znotraj skupine izračunala novo vrednost slikovne točke. Takšen način je na grafičnih karticah zelo učinkovit. Pričakovali bi približno enako učinkovitost istega algoritma na FPGA, vendar ni tako. Zato smo se odločili uporabiti pomikalni register, ki ga prevajalnik VHDL zelo učinkovito implementira na vezje FPGA.

Prvi način smo poimenovali implementacija z lokalnim pomnilnikom, drugega pa implementacija s pomikalnim registrom.

Implementacija z lokalnim pomnilnikom

Pri opisu Sobelovega operatorja (4.1) lahko opazimo, da je za izračun gradienta posamezne slikovne točke potrebnih osem sosednjih točk. Ker pri računanju gradienta sosednjih slikovnih točk velikokrat dostopamo do enakih sosednjih vrednosti je zaželeno, da uporabimo lokalni pomnilnik. Pri tem

je pomembno, da ustvarimo dovolj velike delovne skupine in tako kar najbolj izkoristimo uporabo lokalnega pomnilnika.

Za reševanje celotnega problema ustvarimo niti $h \times w$, kjer h predstavlja višino slike v številu slikovnih točk oziroma število niti v prvi dimenziji, w pa širino slike v številu slikovnih točk oziroma število niti v drugi dimenziji, in jih združimo v lokalni razpon velikosti $lh \times lw$, kjer je lh lokalni razpon v prvi dimenziji in lw lokalni razpon v drugi dimenziji. Za lažje razumevanje si predstavljamo, da so niti združene v delovne skupine v obliki kvadrata, in na to se bomo v nadaljevanju tudi sklicevali.

Vsaka nit znotraj ščepca glede na svoj globalni identifikator kopira vrednost točke v lokalni pomnilnik, in če je nit na robu svoje delovne skupine, kopira še dodatno točko, do katere bo treba dostopati. Če je nit na levem robu skupine, kopira svojega levega soseda, če je na desnem, pa svojega desnega; če je na zgornjem robu, kopira zgornjega, in če je na spodnjem, spodnjega. Če je nit v kotu delovne skupine, temu ustrezno kopira še diagonalno točko v lokalni pomnilnik.

Vsaka nit znotraj skupine nato izračuna gradient za svojo slikovno točko, pri tem pa dostopa le do lokalnega pomnilnika. Rezultat shrani v globalni pomnilnik.

Implementacija s pomikalnim registrom

Pri prejšnji implementaciji Sobelovega filtra je zaradi neposrednega dostopa do globalnega pomnilnika prepustnost manjša, saj posamezne skupine do globalnega spomina dostopajo naključno. Dostop do slikovnih točk znotraj skupine ni zaporeden, saj so niti združene v kvadrat, in tako ščepec dostopa do več vrstic hkrati oziroma pride do več preskakovanja v globalnem pomnilniku.

Ideja, ki je izboljšala prepustnost pomnilnika, je sosednji ali zaporedni dostop do pomnilnika in implementacija s pomikalnim registrom, ki omogoča optimizirano implementacijo na vezju FPGA [14]. Podrobnejši opis optimizacije s pomikalnim registrom si lahko ogledamo v poglavju 5.7. Optimizacije

s kanali nismo uporabili, saj nismo imeli več ščepcev.

Pri tej implementaciji se moramo omejiti na izvajanje ščepca z eno nitjo, ki zaporedno bere točke iz globalnega pomnilnika v pomikalni register ali lokalni pomnilnik, odvisno od velikosti slike. Pomikalni register je enodimenzionalna tabela velikosti $2 \times n + 3$, kjer n predstavlja širino slike oziroma število točk v eni vrstici. Poleg dveh vrstic slike so potrebne še tri dodatne točke, tako da imamo vseh osem sosedov in lahko v vsaki iteraciji izračunamo gradient ene slikovne točke. Število iteracij je enako številu točk v sliki, zaradi potrebne inicializacije pomikalnega registra pa prištejemo še njegovo velikost. Prvih $2 \times n + 3$ iteracij se inicializirajo vrednosti v pomikalnem registru na 0, nato pa se z vsako iteracijo pomaknejo za eno mesto v desno. Nato ščepec prebere slikovno točko iz globalnega pomnilnika in jo zapiše na začetek pomikalnega registra. Z danimi koeficienti (4.1) se izračuna gradient točke po osi x in y in magnituda (4.2). Na koncu se rezultat shrani v globalni pomnilnik.

Slabost te implementacije je konstantna velikost pomikalnega registra, zato je za slike različnih širin treba prevesti ščepce z različnimi in vnaprej definiranimi vrednostmi.

4.5 Rezanje šivov

Rezanje šivov je algoritem za spreminjanje velikosti slike, za njeno povečanje oziroma zmanjšanje, pri tem da se upošteva vsebina slike. Pri določanju vsebine slike se algoritem zanaša na energijo posamezne slikovne točke, ki jo, odvisno od implementacije, izračuna z določeno energetsko funkcijo. Za razliko od običajnega prilagajanja velikosti slike z enostavnim rezanjem slike (angl. crop), pri katerem se iz slike izrežejo le robne točke in tako ostane le njen del, za katerega pa ni nujno, da vsebuje bistvo slike, algoritem rezanja šivov išče slikovne točke z najmanjšo energijo in z odstranitvijo oziroma dodajanjem le-teh prispeva najmanj k energiji celotne slike [18, 19].

Pri spreminjanju velikosti slike je treba tudi paziti, da v vsaki vrstici ozi-

roma stolpcu odstranimo enako število točk in s tem ohranimo enako število točk v vseh vrsticah in stolpcih. Poleg tega je pomembno še, da se točke med seboj povezujejo in iz slike niso naključno izbrane le z upoštevanjem energetske funkcije, saj bi tako prišlo do deformacije slike. Cilj algoritma je poiskati točke, ki slike ne deformirajo, ampak ohranijo njeno obliko in vsebino. Algoritem išče tako imenovane šive, ki pa jih je treba definirati [19].

Naj bo I slika velikosti $n \times m$ in s_i točka v sliki. Vertikalen šiv je določen kot:

$$s^x = \{s_i^x\}_{i=1}^n = \{(x(i), i)_{i=1}^n, \quad \forall i|x(i) - x(i-1)| \leq 1 \quad ,$$

kjer x predstavlja preslikavo $x : [1, ..., n] \rightarrow [1, ..., m]$. Pri horizontalnem šivu je preslikava y ravno obratna od preslikave x, in sicer $y : [1, ..., m] \rightarrow [1, ..., n]$, in je šiv tako formalno definiran kot:

$$s^y = \{s_j^y\}_{j=1}^m = \{(j, x(j))_{j=1}^m, \quad \forall j|x(j) - x(j-1)| \leq 1 \quad .$$

Glede na določeno energetsko funkcijo e točke v sliki, lahko definiramo ceno posameznega šiva kot

$$E(s) = \sum_{i=1}^{n} e(I(s_i)) \quad ,$$

pri tem pa iščemo šiv z najmanjšim vplivom - optimalen šiv

$$s^* = min_s E(s) = min_s \sum_{i=1}^{n} e(I(s_i)) \quad .$$

Običajno za iskanje optimalnega šiva uporabimo dinamično programiranje.

Algoritem rezanja šivov lahko razdelimo na tri dele, in sicer na izračun energije posamezne slikovne točke, izračun minimalnih kumulativnih energij in določitev optimalnega šiva, ter na povečanje oziroma zmanjšanje slike.

Za izračun energije posamezne slikovne točke lahko uporabimo različne energijske funkcije. Pri naši implementaciji smo uporabili Sobelov filter, kot

smo ga definirali v poglavju 4.4 in implementirali v poglavju 4.4.1.

Izračun minimalnih kumulativnih energij za posamezno točko (i, j) lahko zapišemo kot

$$\begin{gathered} k_e(i,j) = k_e(i,j) + \min(k_e(i-1,j-1), k_e(i-1,j), k_e(i-1,j+1)) \\ k_e(1,j) = e(1,j) \end{gathered},$$

kjer k_e predstavlja kumulativno energijo točke s koordinatama i in j. Za izračun minimalne kumulativne energije točk v vrstici i tako potrebujemo že izračunano vrstico $i-1$. Izračun trenutne vrstice je tako odvisen od izračuna prejšnje. Izračun kumulativnih energij začnemo v prvi vrstici in postopek ponavljamo za vsako vrstico vse do zadnje, n.

Po koraku določanja kumulativnih energij vseh točk je treba poiskati najbolj optimalen šiv. Pri naši implementaciji smo se osredotočili predvsem na iskanje optimalnih vertikalnih šivov. V vrstici n so tako izračunane zadnje kumulativne energijske vrednosti slikovnih točk in med vsemi točkami v vrstici n moramo poiskati tisto z najmanjšo vrednostjo. Ta točka predstavlja začetek šiva in tako z vhodnimi kumulativnimi vrednostmi sestavimo celoten šiv. Točki z najmanjšo kumulativno energijsko vrednostjo v vrstici n poiščemo soseda z najmanjšo vrednostjo v vrstici $n-1$. To ponavljamo vse do prve vrstice.

Po določitvi optimalnega šiva lahko sliko povečamo ali zmanjšamo. Pri naši implementaciji smo uporabili le postopek za zmanjšanje slike, in sicer tako, da smo odstranili vse slikovne točke vertikalnega šiva in ustrezno pomaknili slikovne točke. Za odstranitev k slikovnih točk po širini slike smo celoten postopek rezanja šivov ponovili k-krat.

4.5.1 Implementacija algoritma

Pri implementaciji algoritma smo algoritem razdelili na pet podproblemov in pri prvi implementaciji tudi na pet različnih ščepcev:

- energijska funkcija (Sobelov filter),
- izračun minimalne kumulativne energije,
- minimalna kumulativna energija n vrstice,
- izgradnja šiva,
- odstranitev šiva/zmanjšanje slike.

Implementacija z več nitmi

Pri implementaciji z več nitmi imamo v mislih predvsem uporabo ščepcev z več nitmi oziroma prilagoditev algoritma več-nitnemu izvajanju.

Prvi ščepec izračuna energijo posamezne slikovne točke z uporabo Sobelovega filtra. Implementacija in opis algoritma sta enaka, kot smo ju opisali v poglavju 4.4.1.

Drugi ščepec izračuna minimalne kumulativne energije vseh slikovnih točk v k-ti vrstici. Ker je vsaka vrstica odvisna od rezultatov prejšnje vrstice, ščepec izvršimo n-krat, pri čemer n označuje višino slike. Vsaka nit v delovni skupini kopira svojo slikovno točko in zgornjega soseda v lokalni pomnilnik. Robni niti pa kopirata še eno slikovno točko več, saj je za izračun kumulativne energije potrebna najmanjša vrednost izmed zgornjih treh sosedov. Nato vsaka nit poišče najmanjšo vrednost zgornjega soseda, prišteje energetski vrednosti njene slikovne točke, izračunane s Sobelovim filtrom, in rezultat shrani v globalni pomnilnik. Delovne skupine so v razponu 256 niti v eni dimenziji.

V tretjem ščepcu s pomočjo redukcije poiščemo najmanjšo vrednost kumulativne energije v zadnji, n-ti vrstici. V prvem delu ščepca niti poiščejo najmanjšo vrednost. Med seboj primerjajo element v vrstici, ki sovpada z njihovim indeksom v delovni skupini, z vsemi slikovnimi točkami, ki so odmaknjene od te točke za večkratnik velikosti delovne skupine. Širina slike je najbolj odmaknjena slikovna točka. Tako dobimo tabelo velikosti delovne skupine z najmanjšimi vrednostmi v vrstici; potreben pogoj za redukcijo

je, da je velikost delovne skupine enaka potenci števila 2. Niti nato svoje najmanjše vrednosti shranijo v lokalni pomnilnik in se sinhronizirajo. Po sinhronizaciji le prva polovica niti primerja svojo vrednost slikovne točke z vrednostjo, ki je odmaknjena za $ls/2$, kjer ls predstavlja število vseh vrednosti, še potrebnih za primerjavo. Manjšo vrednost nit shrani v lokalni pomnilnik. V naslednji iteraciji primerja le četrtina niti in tako naprej, vse dokler ne primerja le ena nit in v globalni pomnilnik shrani indeks slikovne točke z najmanjšo vrednostjo.

Kljub temu da izgradnja šiva ni paralelni problem, ga sestavimo v četrtem ščepcu. Delovna nit začne sestavljanje šiva pri najmanjši vrednosti v zadnji vrstici in med tremi zgornjimi sosedami izbere točko z najmanjšo kumulativno vrednostjo ter shrani njen indeks v globalni pomnilnik. Postopek ponavlja, vse dokler ne pride do prve vrstice, in tako sestavi celoten šiv.

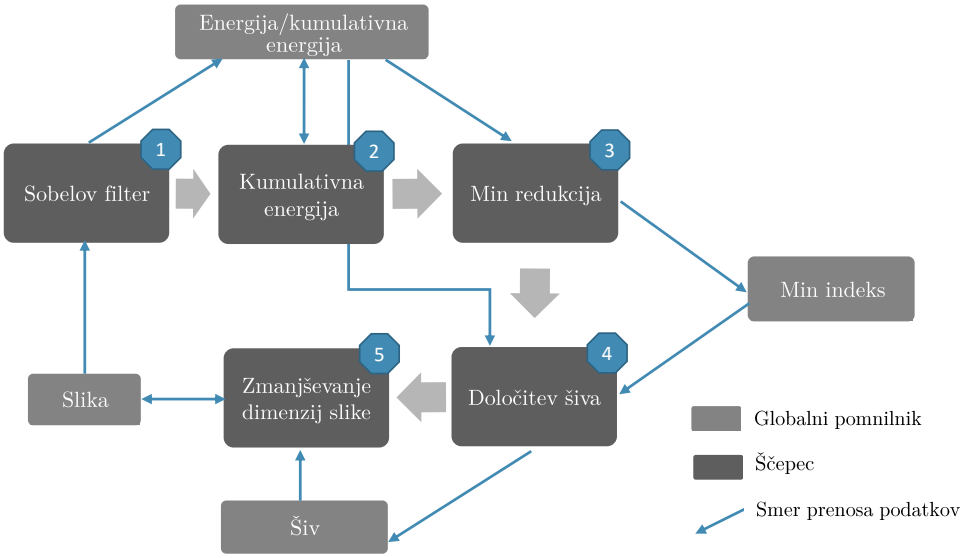
Zadnji ščepec odstrani slikovne točke, ki so del šiva. Vsaka delovna skupina odstrani slikovno točko v svoji vrstici in paralelno zamakne ostale točke na ustrezno mesto. Za boljšo ponazoritev povezovanja med ščepci in izvajanjem algoritma si lahko pogledamo sliko 4.1.

Optimizacije, ki smo jih uporabili, si lahko podrobno pogledamo v poglavju 5.

Implementacija s kanali in pomikalnim registrom

Poleg več nitne implementacije je bila ideja izboljšati izvajanje na FPGA s pomočjo kanalov (angl. channels), kar je Alterina razširitev, podobna že obstoječi implementaciji cevi (angl. pipes) v programskem ogrodju OpenCL. Podrobneje je uporaba kanalov opisana v poglavju 5.8.

Sobelov filter. Dostop do sosednjih točk je veliko hitrejši, zato za računanje energij slikovnih točk uporabimo implementacijo Sobelovega filtra, opisanega v poglavju 4.4.1. Ščepcu dodamo tudi komunikacijo s sosednjim ščepcem preko kanalov. Tako vsako izračunano energijo slikovne točke posreduje naslednjemu ščepcu za izračun kumulativne energije.

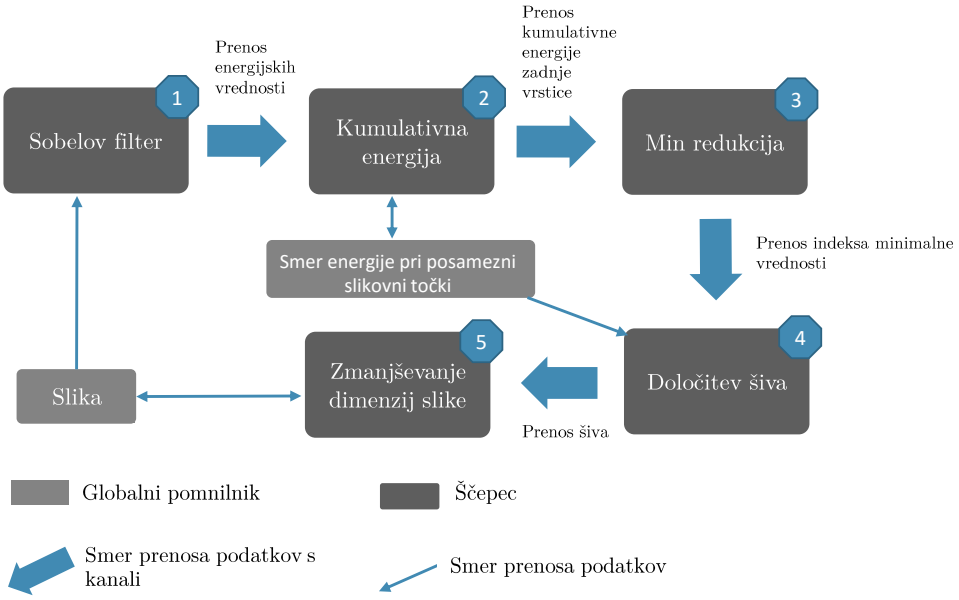


Slika 4.1: Shema prikaza povezovanja med ščepci in globalnim pomnilnikom

Izračun kumulativne energije. Ščepec sprejme slikovno točko iz kanala, jo shrani v pomikalni register in ko ima v pomikalnem registru dovolj točk, začne računati kumulativno energijo. Pri izbiri minimalne kumulativne energije v prejšnji vrstici v globalni pomnilnik zapiše smer za vsako točko z vrednostjo -1, ki označuje levega zgornjega soseda, 0, ki označuje srednjega, ali +1, ki označuje desnega. S tem namesto shranjevanja celotnih kumulativnih vrednosti za vsako slikovno točko shranimo le smer, ki je potrebna za izgradnjo optimalnega šiva. Velikost, ki jo tako zavzame kumulativna energija slikovne točke v globalnem pomnilniku, je en bajt. Pri določanju optimalnega šiva sta tako časovno zahtevna le dostop do globalnega pomnilnika in operacija seštevanja, brez nepotrebnega primerjanja. Ščepec v zadnji vrstici slike poišče najmanjšo kumulativno energijo in indeks te točke posreduje ščepcu za določitev optimalnega šiva.

Določitev optimalnega šiva. Ščepec za določitev optimalnega šiva dobi kot vhodni podatek začetni indeks šiva. Z dostopom do globalnega pomnilnika, kjer so shranjene smeri za vsako slikovno točko, prišteje smer tre-

nutnemu indeksu in tako dobi naslednjo točko, ki je del optimalnega šiva. Ščepec ponavlja postopek vse do prve vrstice oziroma do določitve optimalnega šiva v celoti. Vsak indeks točke, ki ga izračuna, preko kanala posreduje naprej.

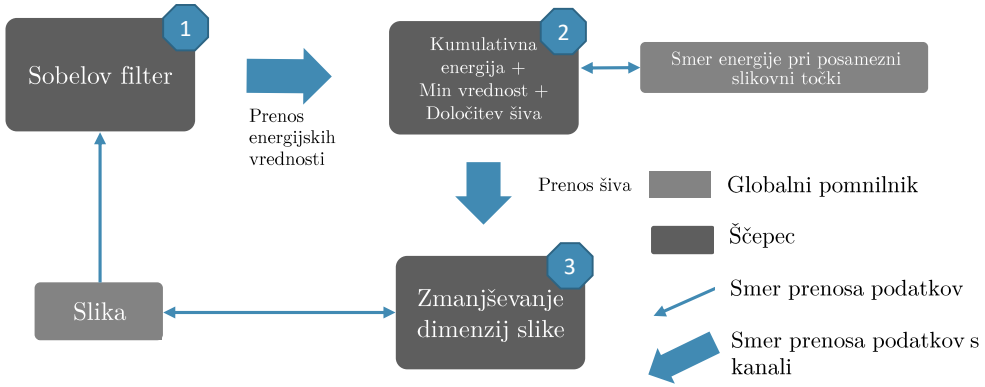


Slika 4.2: Shema prikaza povezovanja med ščepci z uporabo kanalov

Odstranitev šiva. Ščepec za odstranitev šiva ima n niti, kjer n predstavlja višino slike. V kanal se po vrsti zapisujejo indeksi točk šiva od spodnje vrstice navzgor, in tako je treba tudi po vrsti vzeti indekse šiva iz kanala. Zato je vrstni red izvajanja niti pomemben, in ga uveljavi razširitev s kanali. Vsaka nit preko kanala sprejme indeks slikovne točke, ki jo je treba odstraniti v svoji vrstici. Ostale desno postavljene slikovne točke pomakne za eno točko v levo. Slika je tako za eno slikovno točko ožja.

Postopek ponavljamo, dokler nismo zadovoljni s širino slike. Prikaz prenosa podatkov med ščepci s pomočjo kanalov si lahko ogledamo na sliki 4.2

zgoraj. Za razliko od slike 4.1 lahko tu opazimo, da imamo manj prenosa podatkov med ščepci in globalnim pomnilnikom. Izračunane vrednosti si ščepci med seboj posredujejo preko kanalov in tako je izvajanje bolj optimalno. Pri implementaciji rezanja šivov 4.3 pa smo še dodatno pohitrili izvajanje z združitvijo treh ščepcev.



Slika 4.3: Končna optimizirana verzija, prikaz povezovanja med ščepci s pomočjo kanalov

Poglavje 5

Optimizacije

5.1 Manjši podatkovni tipi

Pri podajanju argumentov ščepcu in deklaraciji spremenljivk znotraj ščepca lahko privarčujemo razmeroma veliko logičnih gradnikov že samo z uporabo manjših in ustreznejših podatkovnih tipov. Za vsako spremenljivko glede na možne vrednosti izberemo najustreznejši tip.

5.2 Določitev velikosti lokalnega pomnilnika v ščepcu

Pri uporabi lokalnega pomnilnika v ščepcu prevajalnik privzeto rezervira 16 kB spomina, kar predstavlja velik del logičnih gradnikov na FPGA. Za omejitev in natančno določitev velikosti lokalnega pomnilnika atributu dodamo local_mem_size(N), pri čemer je pogoj, da je N potenca števila 2. Primer uporabe atributa, kjer je velikost lokalnega pomnilnika 4 KB, lahko vidimo v izseku 5.1.

Izsek 5.1: Primer implementacije ščepca z omejitvijo velikosti lokalnega pomnilnika

```
__kernel
void velikost_lokalnega_pomnilnika (
        __attribute__((local_mem_size(4096)))
          __local float * A)
{
  ...
}
```

5.3 Poravnan medpomnilnik (DMA)

Pri gostiteljskem dodeljevanju vmesnega pomnilnika je zaželeno, da so naslovi operandov v vmesnem pomnilniku poravnani. S tem omogočimo prenos podatkov med FPGA in gostiteljem preko neposrednega dostopa do pomnilnika (angl. direct memory access, DMA), ki je bolj učinkovit.

5.4 Zahtevano število niti delovne skupine

Določitev števila niti v ščepcu omogoča, da Alterin prevajalnik izvede agresivno optimizacijo in tako zmanjša porabo sredstev brez uporabe dodatne logike. Ščepcu je treba dodati atribut reqd_workgroup_size(N), kjer se mora število N ujemati z velikostjo delovne skupine, ki jo določi gostitelj. Kadar atribut ni podan in uporabimo pregrado za sinhronizacijo niti znotraj delovne skupine, prevajalnik predpostavi, da je delovna skupina velikosti 256.

5.5 Vektorizacija

Vektorizacija nam omogoča večjo prepustnost ščepcev. Več niti znotraj delovne skupine izvede en ukaz z različnimi toki podatkov (angl. single instruction multiple data, SIMD). Ščepcu je treba dodati atribut num_simd_work_items(I),

kjer I označuje velikost vektorja, nad katerim bo ščepec izvedel določeno operacijo. Pogoj za uporabo omenjenega atributa je uporaba atributa za zahtevano število niti delovne skupine 5.4, kjer mora biti število niti deljivo z I oziroma z delom, ki ga opravi ščepec z enim ukazom.

5.6 Razvoj zanke

Pri optimizaciji ščepcev je zaželeno razviti zanke z uporabo direktive *#pragma unroll N* nad for zanko. Število N pri direktivi predstavlja število iteracij, ki naj jih prevajalnik razvije. Kadar direktivi števila ne podamo, bo prevajalnik poskušal razviti celotno zanko.

Z razvojem zank naj bi izboljšali prepustnost ščepca z več paralelnimi operacijami, z večjo prepustnostjo pomnilnika in z večjim številom operacij v eni urini periodi. Prevajalnik za razvoj zank porabi več logičnih enot na FPGA, kot bi jih sicer.

5.7 Uporaba pomikalnega registra

Pri pogostem dostopu do globalnega pomnilnika pride do številnih zakasnitev in odvisnosti. Velikokrat je spremenljivka odvisna od naslednjega dostopa v globalni pomnilnik, ki je zamuden, onemogoči hitro izvajanje in tako zmanjša prepustnost. Namesto ene urine periode za izračun rezultata je na primer potrebnih sedem urinih period. Da bi se izognili odvisnostim v zankah in prevajalniku omogočili sintetiziranje vezja v pravi cevovodni obliki, je zaželeno uporabiti pomikalni register. Z njim shranimo več operandov iz globalnega pomnilnika s sosednim dostopom, nad katerimi se izvedejo določene operacije, ki pa se zaradi dostopa v zasebni pomnilnik, v pomikalni register, izvedejo mnogokrat hitreje.

5.8 Kanali

Kanale navadno uporabljamo, kadar imamo več ščepcev in lahko vlogo enega ščepca predstavimo kot proizvajalca, drugega pa kot porabnika. Pri takšni predstavitvi lahko namesto pisanja v globalni pomnilnik in branja iz njega uporabimo kar neposredno komunikacijo med ščepci brez koordinacije podatkov na gostitelju. Kanale tako uporabljamo za komunikacijo in sinhronizacijo med ščepci z visoko učinkovitostjo in nizko zakasnitvijo.

Za boljše razumevanje uporabe kanalov si lahko pogledamo primer algoritma v izsekih 3 in 4.

Algoritem 3 Primer uporabe kanala med ščepcem za izračun energije (Sobel) in ščepcem za izračun kumulativne energije

```
1:  STEVILO_TOCK ← KONSTANTA  ▷ Predstavlja širino slike
2:  # pragma OPENCL EXTENSION cl_altera_channels enable
3:  channel int kanal_rezanje_siv
4:  channel int kanal_min_redukcija   ▷ Definiranje kanalov z določenim podatkovnim tipom elementa
5:  function SobelFilter(slika, velikost_slike)
6:      i ← −(2 * STEVILO_TOCK + 3)
7:      pom_reg[2 * STEVILO_TOCK + 3]  ▷ Pomikalni register velikosti dveh vrstic in treh dodatnih točk
8:      while i! = velikost_slike do
9:          ...   ▷ Izračunaj energijo ene slikovne točke
10:         if i >= 0 then   ▷ Energijo posredujemo naprej
11:             write_channel_altera(kanal_rezanje_siv,
                    energija_slikovne_tocke)
12:         end if
13:         i ← i + 1
14:     end while
15: end function
```

Kanali so, za razliko od cevi, blokirajoči klici. Ko je kanal poln, se izvajanje ščepca ustavi, dokler porabnik ne vzame iz kanala vsaj enega elementa. Ob definiranju kanala določimo vrsto in velikost elementa z enim od osnovnih podatkovnih tipov. Kanali delujejo po principu FIFO (angl. first in first

out). Podatki so med delovnimi skupinami in različnimi klici ščepcev konsistentni. Specifikacija OpenCL ne določa vrstnega reda izvajanja niti, vendar za potrebe konsistentnosti programski vmesnik Altera SDK za OpenCL to uveljavi. Izvajanje ščepcev tako poteka v določenem vrstnem redu, in sicer se delovne skupine z nižjim indeksom izvedejo najprej, nato niti z najnižjim indeksom v tretji dimenziji, nato tiste z najnižjim indeksom v drugi dimenziji in na koncu še niti z najnižjim indeksom v prvi dimenziji.

Algoritem 4 Nadaljevanje primera uporabe kanalov, izračun kumulativne energije

```
16: function KumulativnaEnergija(velikost_slike)
17:     i ← 0
18:     pom_reg[STEVILO_TOCK]                ▷ Pomikalni register velikosti
                                               ene vrstice in ene dodatne
                                               točke
19:     while i < velikost_slike do
20:         j ← STEVILO_TOCK + 1
21:         for j > 0; j ← j − 1 do
22:             pom_reg[j] ← pom_reg[j − 1]  ▷ Zamakni pomikalni register
23:         end for                          ▷ Branje iz kanala in zapis v
                                               pomikalni register
24:         pom_reg[0] ← read_channel_altera(kanal_rezanje_siv)
25:         ...                              ▷ Izračunaj kumulativno ener-
                                               gijo slikovne točke z inde-
                                               ksom STEVILO_TOCK v
                                               pomikalnem registru
26:         if zadnja_vrstica = TRUE then
27:             write_channel_altera(kanal_min_redukcija, tocka)
28:         end if
29:         i ← i + 1
30:     end while
31: end function
```

Poglavje 6

Rezultati

6.1 Nenatančni množilnik

Nenatančni množilnik smo implementirali s programskim orodjem OpenCL in z nizkonivojskim jezikom VHDL. Med implementacijami smo nato primerjali število potrebnih gradnikov za sintezo logike na vezje FPGA.

Za izgradnjo ščepcev na vezje FPGA se porabi razmeroma malo gradnikov, za celotno logiko izvajanja ščepcev pa se jih porabi več, saj je treba ustvariti povezave med pomnilnikom in vezjem FPGA, povezave med FPGA in CPE in dodatno logiko za nadzor in izvajanje ščepcev. To lahko vidimo v tabeli 6.1.

Razlika med implementacijo z VHDL in OpenCL je predvsem v tem, da pri implementaciji VHDL nismo implementirali logike za povezovanje s CPE in pomnilnikom, temveč smo brali vhode iz stikal in prikazali rezultat z lučkami LED. Poleg drugačne logike prikazovanja rezultatov pa se pojavi tudi težava zaradi abstrakcije problema, saj je težko implementirati algoritem s ščepci in pričakovati enako sintezo logike na vezju FPGA kot s programskim jezikom VHDL.

Poskušali smo tudi razbrati pretvorbo logike iz ščepcev v VHDL, vendar je prevajalnik za potrebe optimizacije ustvaril precej neberljivo kodo in tako nam ni uspelo najti podobnosti z našo implementacijo v VHDL. Poleg logike

Tabela 6.1: Primerjava porabe gradnikov med različnimi implementacijami

Gradnik	Implementacija VHDL	OpenCL celotna struktura	Ščepec OpenCL
ALM (32075)	66 (<1 %)	3528 (11 %)	865 (2,7 %)
Registri (128300)	134 (<1 %)	3207 (5 %)	2058 (1,6 %)
Spominski bloki (397)	0	16 (4 %)	0
DSP bloki (87)	0	0	0

za implementacijo nenatančnega množilnika je prevajalnik ustvaril entitete za različne pomnilnike, dostop in sinhronizacijo globalnega pomnilnika in entiteto za upravljanje delavcev.

6.2 Matrično množenje

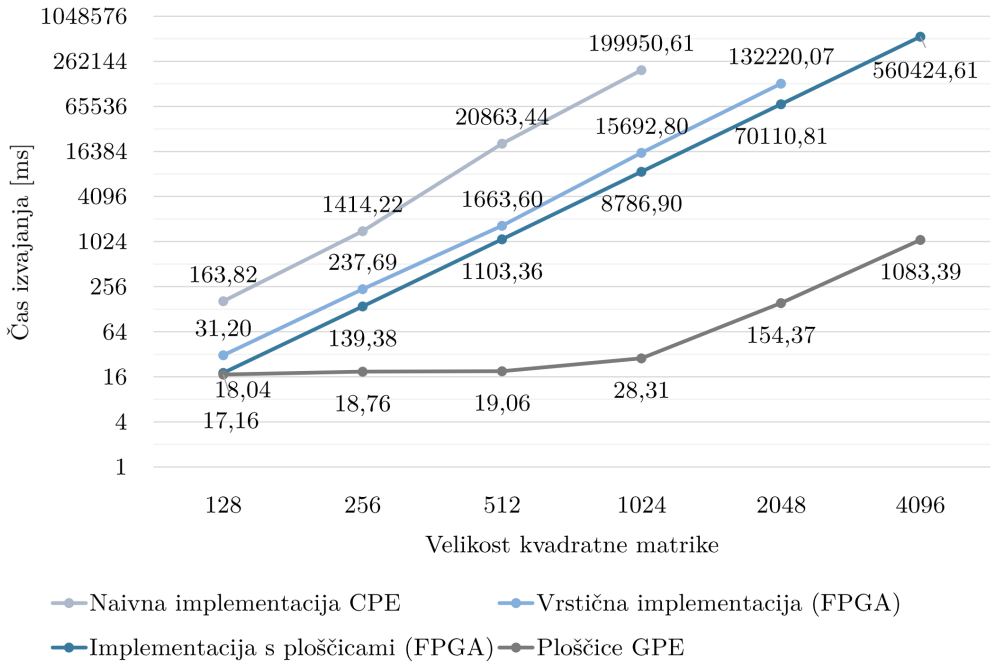
6.2.1 Primerjava implementacij pred optimizacijo

Najprej smo testirali ne-optimizirani implementaciji obeh algoritmov, tako vrstične kot implementacije s ploščicami, na GPE in ploščici FPGA. Za množenje smo izbrali kvadratni matriki velikosti 1024×1024, ki sta vsebovali števila z enojno natančnostjo, ali v programskem jeziku $C++$ tipa float.

Z velikostjo matrik se čas izvajanja eksponentno povečuje. Za boljši prikaz smo na grafu (slika 6.1) uporabili logaritemsko skalo z osnovo 2.

Kot lahko opazimo na zgornjem grafu je izvajanja ščepca na GPE za matrike večjih dimenzij precej hitrejše od izvajanja na vezju FPGA. Pri majhnih dimenzijah pa se FPGA z izvajanjem zelo približa GPE, saj sta prenos majhnih matrik na GPE in izkoristek vseh procesnih enot na grafični kartici premajhna, da bi lahko dosegli optimalne rezultate. Za majhne matrike je izvajanje ščepca na vezju FPGA učinkovito ravno zaradi nasprotnega razloga kot na GPE, saj je prenos razmeroma hiter zaradi majhnega števila elementov.

Opazimo lahko tudi, da se implementacija množenja z vrsticami izvaja



Slika 6.1: Graf časov izvajanja pred optimizacijo algoritmov glede na velikost kvadratne matrike

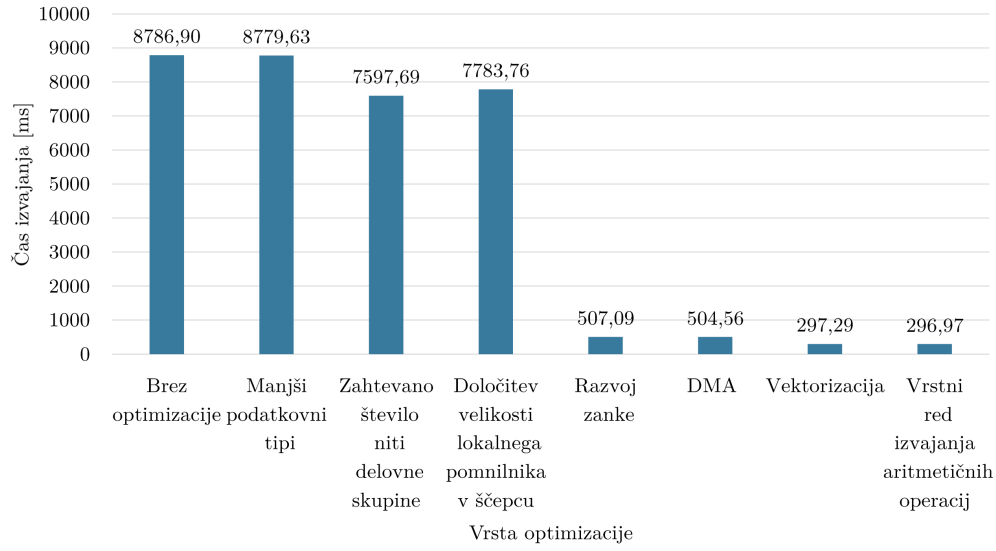
počasneje kot tista s ploščicami, vendar se čas izvajanja pri obeh z velikostjo problema povečuje enakomerno hitro, in to veliko bolj kot izvajanje na GPE, kjer se čas izvajanja povečuje linearno počasi na začetku, pri večji problemih pa se približa eksponentnemu naraščanju.

6.2.2 Vpliv optimizacij na izvajanje

V prejšnjem podpoglavju smo opazili, da se algoritem s ploščicami obnaša bolje kot implementacija z vrstičnim množenjem, zato smo za nadaljnje optimizacije uporabili samo implementacijo s ploščicami. Za testiranje smo, enako kot zgoraj, uporabili kvadratni matriki velikosti 1024×1024, ki sta vsebovali števila z enojno natančnostjo.

Pri optimizaciji algoritma s ploščicami smo uporabili vse optimizacije, opisane v poglavju 5, brez uporabe pomikalnega registra in kanalov. Algo-

ritem smo postopoma nadgrajevali in merili čas izvajanja. Vsaka naslednja optimizacija algoritma je vsebovala vse predhodne optimizacije. Kakšen je čas izvajanja algoritma pri določeni novi optimizaciji lahko vidimo na sliki 6.2



Slika 6.2: Graf časov izvajanja pri dodajanju različnih optimizacij; postopek dodajanja od leve proti desni

Prva optimizacija, pri kateri smo določili ustreznejše podatkovne tipe za spremenljivke in argumente v ščepcih, ne prinese večjih sprememb v času izvajanja, le zmanjša porabo logičnih gradnikov. Pri drugi optimizaciji z določitvijo zahtevanega števila niti pa se izvajanje občutno pohitri, saj prevajalnik optimizira sinhronizacijo niti na podano število. Pri naslednji optimizaciji z določitvijo velikosti lokalnega pomnilnika glede na število niti prihranimo veliko logičnih gradnikov za nadaljnje optimizacije. Pred tem smo porabili 78% vseh spominskih blokov, po tem pa le 31%. Razlog za slabše izvajanje po določitvi velikosti lokalnega pomnilnika ni povsem jasen, obstaja pa možnost, da prevajalnik poskuša optimizirati izvajanje z manjšim številom gradnikov in mu to ne uspe najbolje.

Naslednja optimizacija algoritma z razvojem zank je prelomna. S tem

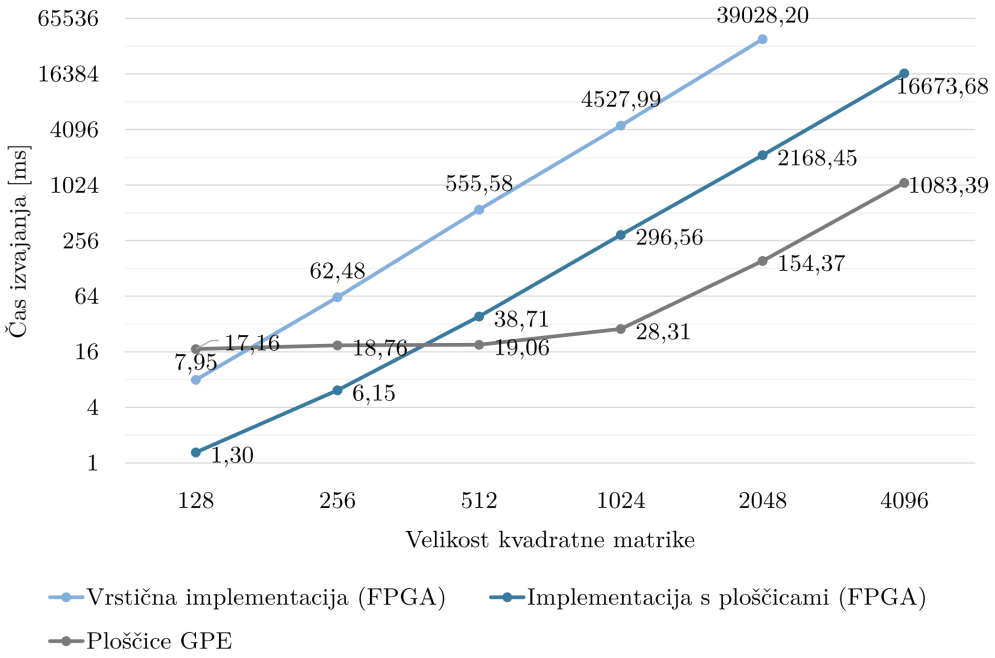
odpravimo odvisnost trenutne iteracije od prejšnje, in tako lahko prevajalnik paralelno sintetizira vse operacije množenja in na koncu rezultate tudi sešteje. Pohitritev je skoraj šestnajstkratna, kar ustreza številu operacij množenja in seštevanja pri posamezni niti.

Pri peti optimizaciji na gostitelju poravnamo podatke in tako omogočimo njihov boljši prenos na FPGA. Direktivo za določitev vektorizacije na posameznem ščepcu omogoča vektorsko izvajanje operacij nad operandi in večjo prepustnost. Pri določitvi vektorizacije velikosti dve dosežemo, kot smo pričakovali, skoraj dvakratno pohitritev, vendar pa se pri poskusu uporabe širših vektorjev izvajanje ščepcev zaradi prevelike obremenitve niti poveča. Izvajanje je še zmeraj hitrejše kot brez vektorizacije. Za najbolj optimalno se tako izkaže vektorizacija z vektorji z dvema elementoma. Ukaz $--fp-relaxed$, opisan v poglavju 3.4.2, pa z bolj "sproščenim" vrstnim redom izvajanja aritmetičnih operacij oziroma zaradi krajšega cevovoda pohitri izvajanje ščepcev.

6.2.3 Primerjava implementacij po optimizaciji

Implementacijo matričnega množenja s ploščicami in vrstično implementacijo smo optimizirali in znova merili čas izvajanja glede na različne velikosti matrik. Implementacije algoritma za GPE nismo posebej optimizirali. Kot lahko opazimo na grafu 6.3, smo pri obeh različicah za FPGA dosegli večkratno pohitritev. V povprečju smo pri vrstični implementaciji dosegli faktor pohitritve 3,5, pri implementaciji s ploščicami pa kar 25,4 v primerjavi z ne-optimiziranimi različicami.

Na sliki 6.1 opazimo, da sta oba algoritma na vezju FPGA pri majhnih matrikah skoraj tako hitra kot pri izvajanju implementacije s ploščicami na GPE. Z dodanimi optimizacijami smo dosegli večjo prepustnost ščepcev in tako prehiteli izvajanje algoritma na GPE.



Slika 6.3: Graf časov izvajanja po optimizaciji algoritmov glede na velikost kvadratne matrike

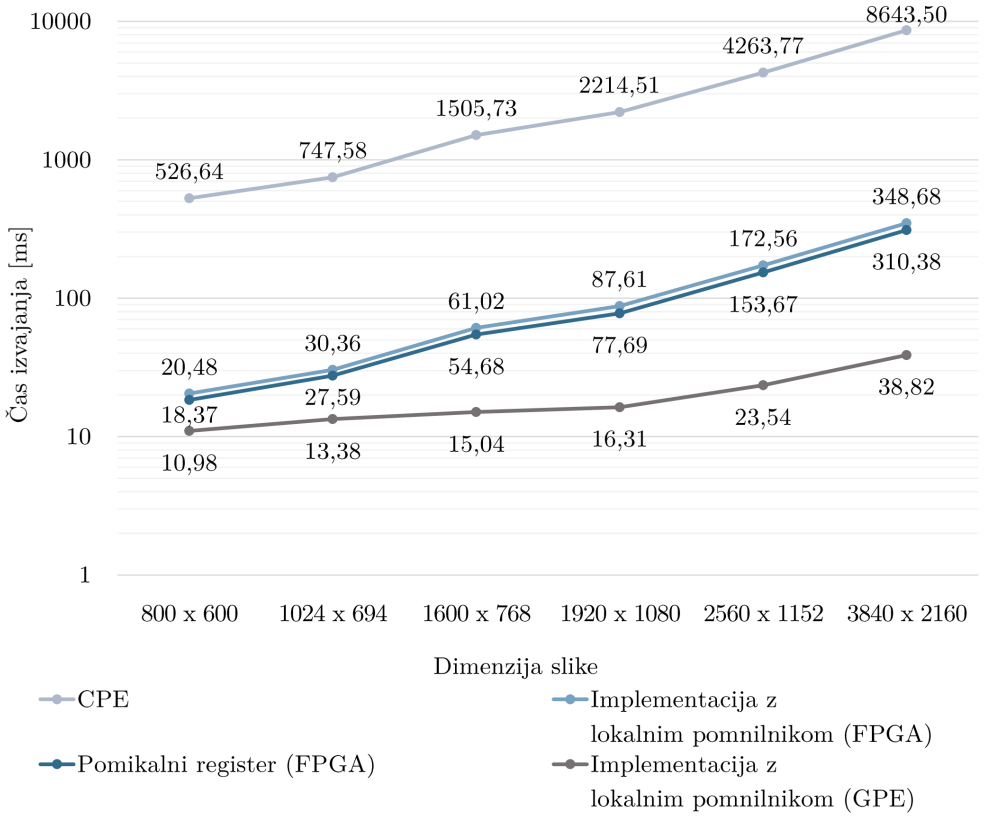
6.3 Sobelov filter

Pri testiranju Sobelovega filtra na vezju FPGA, implementiranega s programskim ogrodjem OpenCL, sta nas zanimala predvsem čas izvajanja ščepcev pri različnih velikostih slik in primerjava med potrebnim časom za zaključitev dela ščepca in časom režije prenosa podatkov in upravljanjem z delavci.

Za potrebe testiranja smo uporabili slike različnih velikosti. Uporabljali smo le sivine slik v datotečnem formatu pgm. Zanimala nas je tudi optimizacija oziroma implementacija s pomikalnim registrom, ki naj bi bila na vezju FPGA učinkovito realizirana.

Na sliki 6.4 lahko opazimo, da je implementacija na GPE najučinkovitejša, implementaciji ščepcev za FPGA pa sta slabši. Različni implementaciji za vezje FPGA se izvajata skoraj enako hitro, vendar je pomikalni register vseeno prinesel nekaj pohitritve. Implementacija s pomikalnim registrom pri-

nese faktor pohitritve 1,1. Glede na velikost slike se razmeroma enakomerno povečuje čas izvajanja ščepcev na vseh napravah.

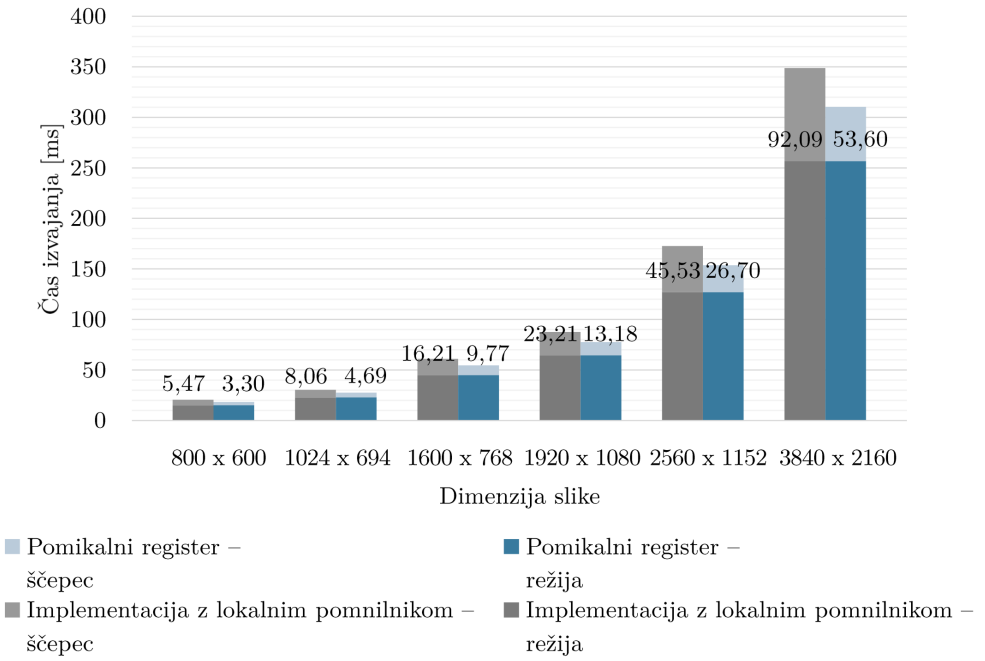


Slika 6.4: Graf časov izvajanja algoritmov glede na velikost slike

S pretvorbo implementacije Sobelovega filtra z uporabo lokalnega pomnilnika, ki nima omejitev glede velikosti slik, v implementacijo s pomikalnim registrom, ki ima določeno širino slike, smo dosegli zmanjšanje števila porabljenih elementov ALM z 11529 ali 36% vseh na 4816 oziroma 15% vseh elementov ALM na vezju FPGA. Za dostop do lokalnega pomnilnika in za upravljanje pomnilnika je bila potrebna dodatna logika, ki pa je povečala število potrebnih gradnikov, ti pa so vplivali na frekvenco ure. Za implementacijo s pomikalnim registrom so potrebni le privatni registri, do katerih je dostop najhitrejši. Poleg števila potrebnih elementov ALM smo zmanjšali

tudi porabo blokov DSP.

Pri sintezi logike ščepcev je tako prevajalniku uspelo optimizirati vezje FPGA in uporabiti uro z višjo frekvenco. Pri implementaciji z lokalnim pomnilnikom se je ščepec na vezju FPGA izvajal pri frekvenci ure 131,3 MHz, pri implementaciji s pomikalnim registrom pa s frekvenco 155,78 MHz. Na grafu 6.5 so vidne tudi izboljšave v času izvajanja ščepcev. Števila nad stolpci označujejo le čas izvajanja ščepcev.



Slika 6.5: Čas izvajanja ščepca in čas, potreben za prenos, režijo, glede na velikost slik; primerjava med bločno implementacijo in implementacijo s pomikalnim registrom

Opazimo lahko še, da je čas, potreben za prenos podatkov, slike in režijo delavcev, pri obeh implementacijah skoraj identičen. Sinteza je pri obeh implementacijah izkoristila največjo možno hitrost prenosa podatkov iz globalnega pomnilnika v lokalni oziroma privatni pomnilnik. Izvajanje se razlikuje le v delu ščepcev ali ščepca in zakasnitvi pri dostopu do različnih pomnilnikov.

6.4 Rezanje šivov

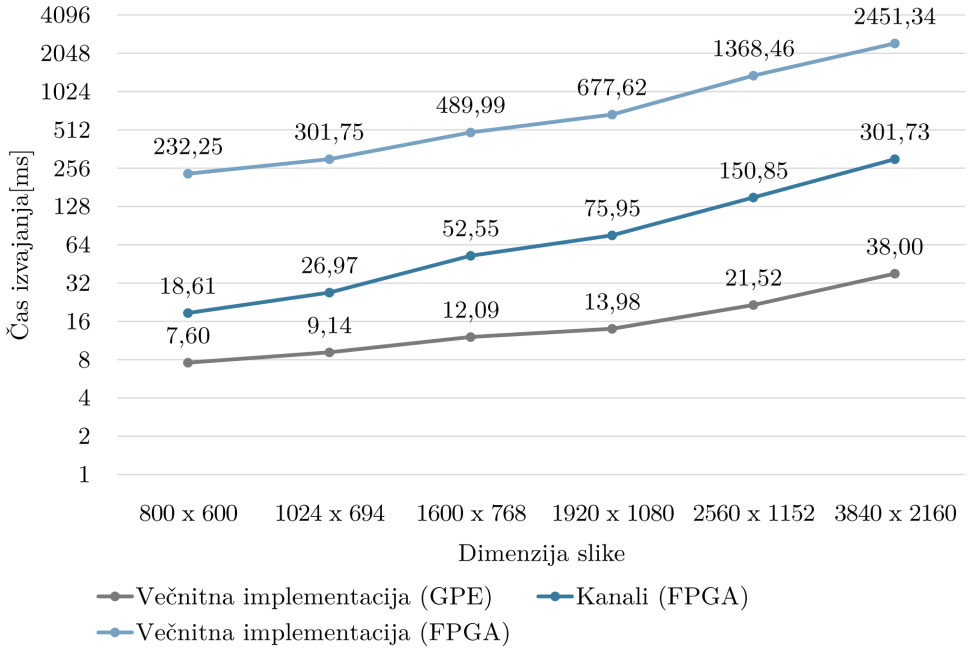
Pri testiranju učinkovitosti paralelizacije na vezju FPGA smo želeli testirati algoritem z večjo kompleksnostjo. Za to nalogo se je algoritem rezanja šivov izkazal kot ustrezen, saj je predstavljal nadgradnjo Sobelovega filtra, hkrati pa je za realizacijo potrebnih več ščepcev hkrati.

Slika 6.6 prikazuje čas odstranjevanja enega šiva iz slike pri različnih implementacijah na različnih arhitekturah. Slike, ki smo jih uporabili, so bile vsebinsko podobne, vendar ne povsem enake. Neenakost slik že predstavlja problem, saj je za odstranitev enega šiva potrebna druga lokacija in tako je treba opraviti več dela. Če bi se slike razlikovale tudi vsebinsko, bi to še slabše vplivalo na ustrezno primerjavo časa izvajanja pri slikah različnih velikosti. Os Y je na sliki prikazana v logaritemski skali, saj vrednosti implementacij na vezju FPGA hitro pobegnejo čez mejo.

Najprej smo testirali implementacijo z več nitmi (4.5.1) na GPE in skoraj z enakimi nastavitvami tudi na vezju FPGA. Rezultate si lahko ogledamo na sliki 6.6. Opazimo lahko, da ti niso bili zadovoljivi. Čas izvajanja implementacije z več nitmi na vezju FPGA se je v povprečju izvajal 46-krat slabše kot na GPE. Potrebni sta bili pohitritev in optimizacija za arhitekturo vezja. Pri tem smo uporabili vse optimizacije, ki smo jih že uporabili pri drugih rešitvah, in zaradi implementacije z več ščepci smo imeli možnost testirati tudi kanale, Alterino razširitev programskega ogrodja OpenCL.

Kanali so prinesli zelo zadovoljive rezultate. V primerjavi z več nitno implementacijo na vezju FPGA je implementacija s kanali prinesla faktor pohitritve 9,9. Čas izvajanja z velikostjo problema narašča hitreje kot na grafični kartici, kar pa je posledica prenosa podatkov in velikosti pomikalnega registra. Če je širina slike večja od največje dovoljene vrednosti za privatni pomnilnik, se bo pomikalni register realiziral z uporabo lokalnega pomnilnika in s tem vplival na hitrost izvajanja ščepca. GPE ustrezajo problemi večjih dimenzij, saj takrat pride do večjega izkoristka vseh procesnih enot. Kljub temu da nam je uspelo močno izboljšati algoritem, nismo mogli prehiteti grafične kartice.

Implementacija s kanali je zmanjšala tudi porabo elementov ALM na vezju FPGA. Porabo nam je s 83% vseh porabljenih elementov ALM pri večnitni implementaciji uspelo znižati na 57%, pri slikah velikosti 3849×2160 pa na 61%. Pri tem se je tudi največja frekvenca ure na vezju FPGA v povprečju zvišala za 5,6% oziroma s 124 MHz na 131 MHz.



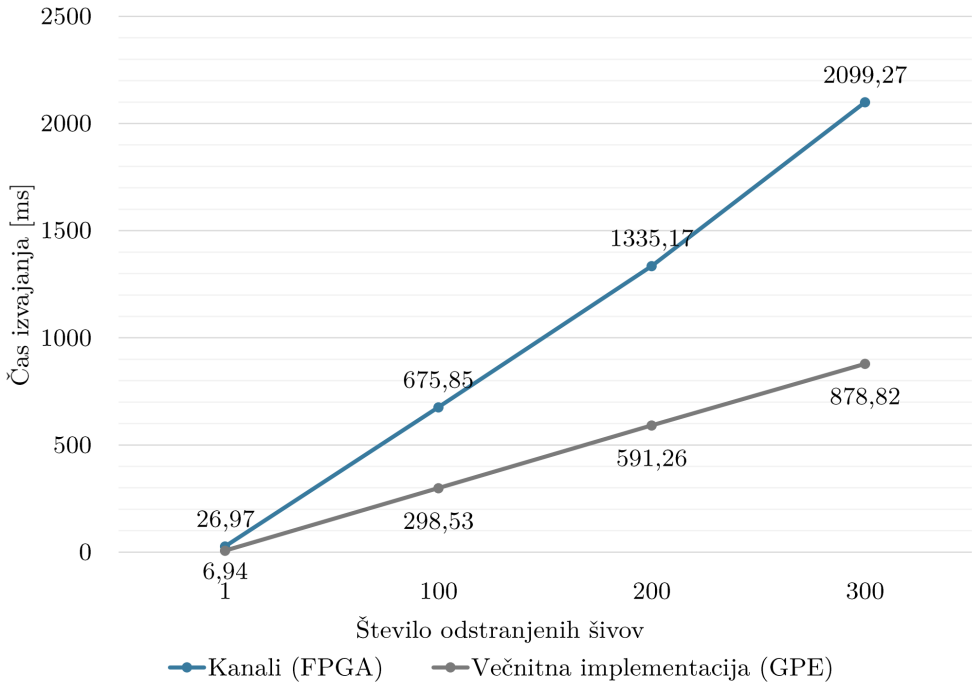
Slika 6.6: Graf časov izvajanja glede na velikost slike, odstranitev enega šiva

Na sliki 6.7 smo primerjali čas izvajanja pri odstranjevanju večjega števila šivov. Primerjali smo le implementacijo s kanali na vezju FPGA in več nitno implementacijo na GPE. Izvajanje smo testirali s sliko velikosti 1024×694.

Kot lahko opazimo, se izvajanje povečuje linearno glede na število odstranjenih šivov. Pri implementaciji s kanali čas izvajanja narašča hitreje, kar je najverjetneje zaradi hitrosti dostopa do globalnega pomnilnika. V tej implementaciji imamo v vsaki iteraciji tri ščepce, ki odstranijo en šiv. Prvi ščepec iz globalnega pomnilnika zaporedno prebere vse slikovne točke, drugi zaporedno zapiše smer minimalne vrednosti za vsako slikovno točko od

druge vrstice navzdol, torej branje iz globalnega pomnilnika. Pri določanju šiva drugi ščepec bere slikovne točke iz globalnega pomnilnika z naključnim dostopom, kar zelo upočasni izvajanje. Tretji ščepec zaporedno prebere le potrebne slikovne točke, desno od šiva v vsaki vrstici, in jih nato zapiše nazaj v globalni pomnilnik. Tako imamo za odstranitev enega šiva en ščepec, ki samo bere, in dva, ki bereta in zapisujeta. Pri odstranitvi večjega števila šivov pride počasni dostop do globalnega pomnilnika še bolj do izraza. Na GPE je dostop do globalnega pomnilnika hitrejši, zato je naraščanje časa izvajanja počasnejše.

Na hitrost naraščanja časa izvajanja lahko vpliva tudi število ščepcev oziroma upravljanje večjega števila ščepcev v vrsti. Pri 300 iteracijah imamo kar 900 ščepcev v vrsti, kar predstavlja veliko režijskih stroškov.



Slika 6.7: Čas izvajanja pri odstranjevanju večjega števila šivov, velikost slike 1024×694

Poglavje 7

Zaključek

V delu smo razvili testne aplikacije za testiranje uporabe programskega ogrodja OpenCL na vezju FPGA. Razvili smo več testnih programov, s katerimi smo testirali različne lastnosti, zmogljivosti vezja in vplive različnih optimizacij na izvajanje ščepcev na vezju FPGA. Odločili smo se za več različnih implementacij, od najbolj splošnih do bolj prilagojenih za vezje FPGA. Te delujejo le na Alterinih programirljivih vezjih, ki podpirajo standard OpenCL.

S pomočjo strojno opisnega jezika VHDL smo implementirali tudi algoritem na vezju FPGA in isti algoritem z uporabo OpenCL poskušali pretvoriti v implementacijo s ščepci. Po pričakovanjih smo ugotovili, da za razvoj aplikacije z uporabo programskega ogrodja OpenCL prihranimo veliko časa, vendar to negativno vpliva na čas izvajanja. Za potrebe implementacije ščepcev, komunikacije med njimi in dodatne logike za upravljanje ščepcev potrebujemo na vezju FPGA večje število logičnih gradnikov. Zaradi tega smo omejeni pri realizaciji večjih aplikacij.

Pri uporabi ukazov za omejevanje števila niti, pri omejevanju velikosti lokalnega pomnilnika in izbiri primernejših podatkovnih tipov za dani problem smo opazili, da lahko privarčujemo ogromno logičnih gradnikov na vezju FPGA. Dodatne direktive, ki jih ponuja Alterina razširitev programskega ogrodja OpenCL, pa lahko omogočijo tudi vzporedno računanje jeder zank oziroma iteracije zank razvijemo v vzporedne iteracije na vezju FPGA. Z

uporabo te optimizacije smo dosegli tudi 16-kratno pohitritev.

Pri testiranju algoritmov smo ugotovili, da na izvajanje ščepcev močno vpliva tudi število dostopov do globalnega pomnilnika, ščepci z uporabo lokalnega in privatnega pomnilnika pa se izvajajo izjemno hitro. Ker je že splošno znano, da je dostop do globalnega pomnilnika počasen, se pri učinkoviti implementaciji ščepcev za izvajanje na vezju FPGA, še bolj omejimo na lokalni in privatni pomnilnik.

Pri implementaciji z več nitmi in pri definiranju problema na več dimenzionalnem razponu bi pričakovali boljše rezultate, vendar ni vedno tako. Osredotočili smo se še na testiranje ščepcev z eno nitjo in implementacijo algoritmov s pomikalnim registrom. Kadar smo zaporedno dostopali do pomnilnika in smo lahko za rešitev problema uporabili pomikalni register, se je izkazalo, da je sinteza algoritma na vezje FPGA učinkovita. Pri uporabi več niti se vsaka nit ne izvaja vzporedno, ampak izkoriščajo cevovod. Uspešno preveden problem v ščepec z uporabo pomikalnega registra lahko prinese pohitritve.

Ugotovili smo tudi, da je pri uporabi več ščepcev hkrati zelo zaželeno uporabiti kanale. To je prav tako razširitev programskega ogrodja OpenCL, ki omogoča prenos podatkov med ščepci. S tem se izognemo počasnemu dostopu do globalnega pomnilnika in pohitrimo izvajanje ščepcev.

Testiranje matričnega množenja na GPE in vezju FPGA je pokazalo, da je na splošno veliko učinkovitejša GPE, ki je občutno zmogljivejša, vendar moramo uporabiti problem ustrezne velikost. Pri majhnih problemih je uspelo optimizirani obliki za vezje FPGA prehiteti ne-optimizirano obliko za GPE. Kljub temu da se je v večini primerov GPE izkazala za učinkovitejši pospeševalnik, smo z rezultati zelo zadovoljni. Pokazali smo, kako močno posamezne optimizacije in prilagoditve ščepcev vplivajo pri dani arhitekturi.

Za nadaljnje delo bi lahko izboljšali dostop do globalnega pomnilnika oziroma poiskali metodo, ki bi najbolj učinkovito prenesla podatke. Možna nadgradnja je tudi testiranje učinkovitosti izvajanja ščepcev glede na porabo električne energije. Čeprav lahko sklepamo, da bi v tem primeru vezje FPGA

prišlo resnično v ospredje, bi bilo to zanimivo preveriti in se o tem povsem prepričati. Vendar pa tega nismo storili, saj bi za točne meritve potrebovali več naprav za merjenje porabe električne energije. V nadaljnjem delu bi bilo zaželeno testirati zmogljivejša vezja FPGA z uporabo OpenCL in jih uporabiti za pohitritev praktičnih aplikacij.

Literatura

[1] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, str. 13–24, 2014.

[2] S. O. Settle, "High-performance dynamic programming on fpgas with opencl," in *Proc. IEEE High Perform. Extreme Comput. Conf.(HPEC)*, str. 1–6, 2013.

[3] D. Chen and D. Singh, "Invited paper: Using opencl to evaluate the efficiency of cpus, gpus and fpgas for information filtering," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, str. 5–12, avg. 2012.

[4] G. Kyriazis, "Heterogeneous system architecture: A technical review," *AMD Fusion Developer Summit*, 2012.

[5] A. Munshi, B. Gaster, T. G. Mattson, and D. Ginsburg, *OpenCL programming guide*. Pearson Education, 2011.

[6] *Cyclone V Device Overview*. Dostopno na: `https://www.altera.com/en_US/pdfs/literature/hb/cyclone-v/`

cv_51001.pdf.
[Dostopano 5.10.2015].

[7] K. O. W. Group *et al.*, "The opencl specification, version 1.2, 15 november 2011," *Cited on pages*, 18. izd., št. 7, p. 30.

[8] *Techpowerup - GIGABYTE HD 7870 WindForce 3X OC*. Dostopno na: https://www.techpowerup.com/gpudb/b468/gigabyte-hd-7870-windforce-3x-oc.html.
[Dostopano 9.3.2016].

[9] J. Bradley, J. Macaulay, A. Noronha, and H. Sethi, *DE1-SoC User Manual*, 2015. Dostopno na: ftp://ftp.altera.com/up/pub/Altera_Material/Boards/DE1-SoC/DE1_SoC_User_Manual.pdf.
[Dostopano 5.3.2016].

[10] *DE1-SoC My first FPGA*, 2013. Dostopno na: http://terasic.yubacollegecompsci.com/resources/My_First_Fpga.pdf.
[Dostopano 9.3.2016].

[11] *DE1-SoC My first HPS*, 2013. Dostopno na: https://rocketboards.org/foswiki/pub/Projects/DE1SOCMyFirstHPS/My_First_HPS.pdf?t=1471001068.
[Dostopano 9.3.2016].

[12] *DE1-SoC My first HPS-FPGA*, 2013. Dostopno na: http://terasic.yubacollegecompsci.com/resources/My_First_HPS-Fpga.pdf.
[Dostopano 9.3.2016].

[13] *Altera SDK for OpenCL Programming Guide*, 2015. Dostopno na: https://www.altera.com/content/dam/altera-www/global/en_US/

pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf. [Dostopano 5.3.2016].

[14] *Altera SDK for OpenCL Best Practices Guide*, 2015. Dostopno na: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_optimization_guide.pdf. [Dostopano 9.3.2016].

[15] U. Lotrič and P. Bulić, "Applicability of approximate multipliers in hardware neural networks," *Neurocomputing*, 96. izd., str. 57–65, 2012.

[16] Z. Babić, A. Avramović, and P. Bulić, "An iterative logarithmic multiplier," *Microprocessors and Microsystems*, 35. izd., št. 1, str. 23–33, 2011.

[17] B. Orel, *Linearna algebra*. 2013.

[18] M. Rubinstein, A. Shamir, and S. Avidan, "Improved seam carving for video retargeting," in *ACM transactions on graphics (TOG)*, 27. izd., str. 16, ACM, 2008.

[19] S. Avidan and A. Shamir, "Seam carving for content-aware image resizing," in *ACM Transactions on graphics (TOG)*, 26. izd., str. 10, ACM, 2007.